

1. Tehnici de programare

1.1. Subprogramele

1.1.1. Definiția subprogramului

Știți deja că **blocul** este unitatea de bază a oricărui program C++ și că este încapsulat într-o instrucțiune compusă, delimitată de caracterele `{ ... }`. El este format din două părți:

- **Partea declarativă** conține definiții de elemente necesare algoritmului pentru rezolvarea problemei: constante (**const**), variabile de memorie și tipuri de date (**typedef**). Definirea lor se face cu ajutorul **instrucțiunilor declarative**.
- **Partea executabilă** sau **partea procedurală** conține instrucțiunile care descriu pașii algoritmului care trebuie implementat pentru rezolvarea problemei. Aceste instrucțiuni se numesc **instrucțiuni imperative**. Ele sunt: **instrucțiunea expresie** (prin care se evaluează o expresie) și **instrucțiunile de control** (prin care se modifică ordinea de execuție a programului). Instrucțiunea expresie prin care se atribuie unei variabile de memorie o valoare se mai numește și **instrucțiune de atribuire**, iar instrucțiunea expresie prin care se cere execuția unui subprogram se mai numește și **instrucțiune procedurală**.

Mai știți că în limbajul C++ blocurile sunt încapsulate în **funcții**, **orice program C++ fiind o colecție de definiții de variabile și funcții**. Funcția este un bloc precedat de un **antet** prin care se precizează numele ei și, dacă este cazul, tipul rezultatului pe care-l întoarce prin chiar numele său și, eventual, parametri de execuție (valori care se transmit blocului și care sunt necesare atunci când se execută blocul):

```
<tip_rezultat> <nume_funcție>(<listă_parametri>)
```

Una dintre funcțiile programului C++ este **funcția rădăcină**. Ea este obligatorie și este primul bloc cu care începe execuția programului. Numele său este **main**. Antetul acestei funcții este:

```
void main()
```

ceea ce semnifică faptul că funcția nu întoarce nici un rezultat (**void**) și nu necesită parametri pentru apelare – parantezele `()` nu conțin listă de parametri.

Studiu de caz

Scop: exemplificarea structurii unui program C++.

Enunțul problemei: Se consideră trei numere naturale, **a**, **b** și **c**. Să se verifice dacă pot forma o progresie aritmetică.

Se vor executa următorii pași:

Pas1. Se ordonează crescător cele trei numere (se schimbă valorile între ele, astfel încât să se respecte relația de ordine $a \leq b \leq c$).

Pas2. Dacă între valorile celor trei variabile există relația $b = (a + c) / 2$, atunci cele trei numere formează o progresie aritmetică.

partea executabilă

```
#include <iostream.h>

void main()  → antetul funcției
{
  int a,b,c,x; }  partea declarativă

  cout<<"a= "; cin>>a;
  cout<<"b= "; cin>>b;
  cout<<"c= "; cin>>c;

  if (a>b) {x=a;
           a=b;
           b=x;}
  if (b>c) {x=b;
           b=c;
           c=x;};
  if (a>b) {x=a;
           a=b;
           b=x;};

  if (b==(a+c)/2.)
    cout<<a<<"", "<<b<<"", "<<c<<"sunt in progresie aritmetica";
  else
    cout<<a<<"", "<<b<<"", "<<c<<"nu sunt in progresie aritmetica";
}
```

Secvență de instrucțiuni care se execută repetat de trei ori, de fiecare dată cu alte date de intrare:

- 1) a și b
- 2) b și c
- 3) a și b



Prin definiție:

Subprogramul este o secvență de instrucțiuni care rezolvă o anumită sarcină și care poate fi descrisă separat de blocul rădăcină și lansată în execuție din cadrul unui bloc, ori de câte ori este nevoie.

În limbajul C++, **subprogramele** se mai numesc și **funcții**.

1.1.1.1. Necesitatea folosirii subprogramelelor

În practica programării pot să apară următoarele cazuri:

- a. **O secvență de instrucțiuni se repetă** de mai multe ori în cadrul unui program (ca în programul C++ din exemplul anterior). Secvența de instrucțiuni care se repetă poate fi implementată sub forma unui subprogram.
- b. **Rezolvarea unei anumite sarcini este necesară în mai multe programe**, ca, de exemplu, diferite operații matematice (extragerea radicalului, extragerea părții întregi sau a părții fracționare dintr-un număr real, ridicarea unui număr la o putere etc.), diferite operații cu tablouri de memorie (crearea, parcurgerea și sortarea tabloului de memorie, ștergerea sau inserarea unui element etc.), diferite operații cu fișiere (deschiderea unui fișier, închiderea unui fișier, testarea sfârșitului de fișier etc.). Secvența de instrucțiuni care rezolvă o anumită sarcină ce poate să apară în mai multe programe poate fi implementată cu ajutorul unui subprogram.
- c. **Orice problemă poate fi descompusă în subprobleme**. Subproblemele în care este descompusă se numesc **module**. Descompunerea poate continua până când se obține

un modul cu rezolvare imediată. Această metodă de rezolvare a unei probleme se numește tehnica **top-down** de proiectare a algoritmilor. Ea este foarte utilă în cazul programelor care trebuie să rezolve probleme complexe (de exemplu, prelucrarea vectorilor). În aceste cazuri se obțin programe foarte mari și complexe. Pentru a obține programe mai simple și mai clare, se poate descompune problema inițială în subprobleme, fiecare subproblemă fiind descrisă printr-un subprogram.

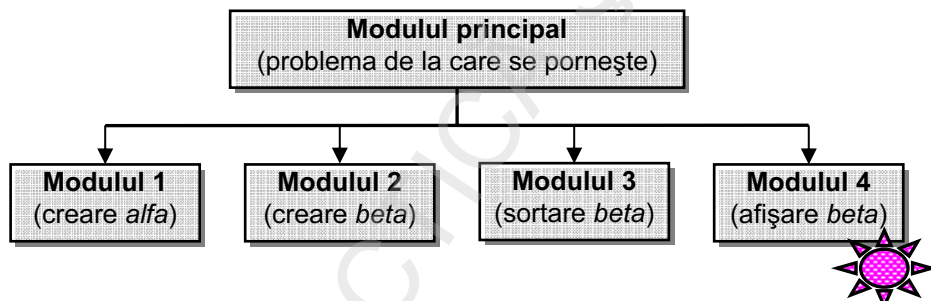
Studiu de caz

Scop: exemplificarea modului în care o problemă poate fi descompusă în subprobleme folosind **tehnica top-down**.

Enunțul problemei: Se introduc de la tastatură mai multe numere întregi, într-un vector **alfa**. Să se transfere în vectorul **beta** elementele pozitive din **alfa** și apoi să se afișeze elementele vectorului **beta**, ordonate crescător.

Problema poate fi împărțită în patru subprobleme (**module**):

- crearea vectorului *alfa*, prin introducerea valorilor de la tastatură;
- crearea vectorului *beta*, prin copierea valorilor pozitive din vectorul *alfa*;
- sortarea vectorului *beta*;
- afișarea elementelor vectorului *beta*.



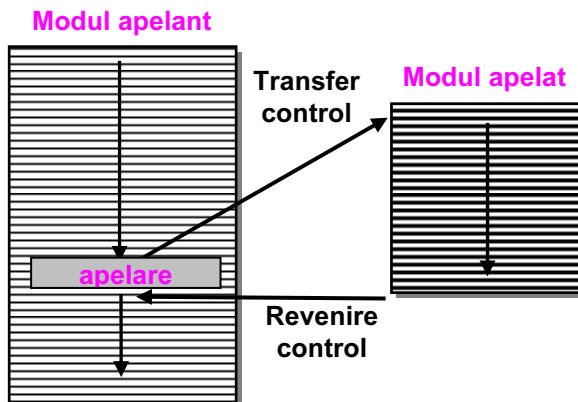
Așadar, în toate cele trei cazuri prezentate, **soluția o reprezintă subprogramele**.

1.1.1.2. Terminologie folosită pentru subprograme

Într-o structură modulară în care fiecare modul este descris printr-un subprogram, modulele se clasifică astfel:

→ **Modul apelant.** Este modulul care, pentru rezolvarea propriei probleme, apelează la alte module, fiecare dintre ele rezolvând o anumită subproblemă. La apelare, el transferă controlul modulului apelat. În exemplul anterior, *Modulul principal* este modulul apelant.

→ **Modul apelat.** Este un modul apelat de un alt modul, pentru a-i rezolva o subproblemă. După ce își termină execuția, el redă controlul modulului apelant. În exemplul anterior, *Modulul 1*, *Modulul 2* și *Modulul 3* sunt module apelate.



Vom considera funcția rădăcină **main()** ca fiind **modulul principal** sau **programul principal**, iar celelalte funcții (module) pe care le vom defini le vom numi **subprograme**.

1.1.1.3. Avantajele folosirii subprogramelor

În practică, pentru rezolvarea unor probleme complexe care ajută la îndeplinirea unor activități, cum sunt de exemplu prelucrările de texte, contabilitatea unei întreprinderi, inventarierea unor depozite de materiale, gestionarea unei biblioteci etc., trebuie să se conceapă programe sofisticate numite **aplicații**. În construirea unei aplicații, folosirea subprogramelor oferă următoarele avantaje:

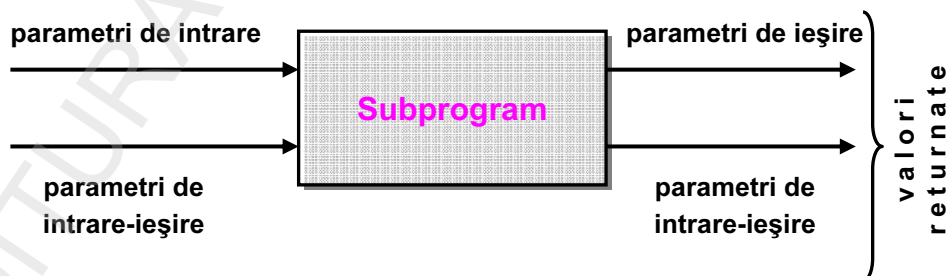
- **Se face economie de memorie internă.** Un grup de instrucțiuni care trebuie să se execute de mai multe ori într-o aplicație (chiar cu date de intrare și de ieșire diferite) se va scrie o singură dată într-un subprogram și se va executa prin apelarea subprogramului ori de câte ori este nevoie.
- **Se favorizează lucrul în echipă** pentru aplicațiile mari. Fiecare programator va putea să scrie mai multe subprograme, independent de ceilalți programatori din echipă. Pentru a realiza subprogramul, este suficient să i se precizeze programatorului specificațiile subprogramului: datele de intrare, datele de ieșire și problema pe care trebuie să o rezolve.
- **Depanarea și actualizarea aplicației se fac mai ușor.** După implementare și intrarea în exploatare curentă, o aplicație poate necesita modificări, ca urmare a schimbării unor cerințe. Este mult mai simplu să se gândească modificarea la nivelul unui subprogram, decât la nivelul întregii aplicații.
- **Crește portabilitatea programelor.** Subprogramele sunt concepute independent de restul aplicației și unele dintre ele pot fi preluate, fără un efort prea mare, și în alte aplicații, în care trebuie să fie rezolvate sarcini similare.

1.1.2. Parametrii de comunicare

Pe de o parte, subprogramul nu este o entitate independentă. El trebuie asamblat în interiorul programului, adică trebuie **stabilite legături între modulul apelant și modulul apelat**.

Pe de altă parte, în procesul de prelucrare dintr-un modul sunt necesare date care trebuie prelucrate (date de intrare) și care uneori trebuie să fie preluate din modulul apelant. La rândul său, în urma prelucrărilor, modulul apelat furnizează rezultate (date de ieșire) către modulul care l-a apelat. Datele care circulă astfel între module se numesc **parametri de comunicare**. Așadar:

Parametrii de comunicare se folosesc pentru a realiza legătura dintre module.



După modul în care intervin în comunicarea cu modulul apelant, parametrii de comunicare se clasifică în:

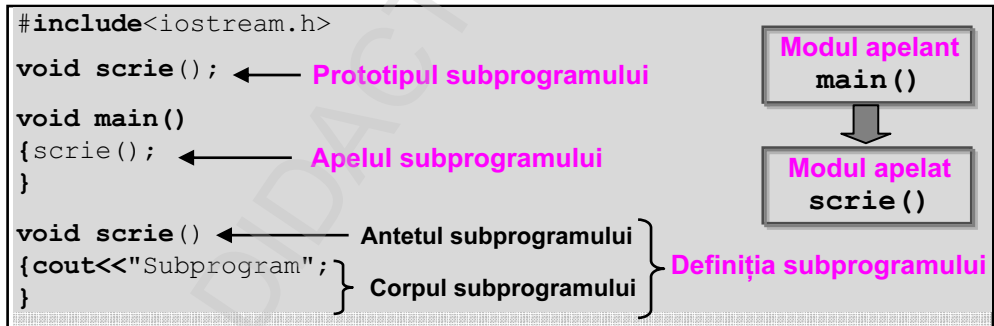
- **Parametrii de intrare.** Sunt date care urmează să fie prelucrate de subprogram (**SP**) și care îi sunt comunicate de către modulul apelant (**P**). Subprogramul le primește în momentul activării: **P** ⇒ **SP**.
- **Parametrii de ieșire.** Sunt rezultate obținute de subprogram în urma prelucrării și pe care le comunică modulului apelant. Modulul apelant le primește după ce subprogramul își termină execuția: **SP** ⇒ **P**.
- **Parametrii de intrare-ieșire.** Sunt date care participă la calculul datelor de ieșire și sunt accesibile atât modulului apelant, cât și modulului apelat. Valoarea lor poate fi modificată atât de subprogram, cât și de modulul apelant. Subprogramul le primește la activare, iar modulul apelant le primește după ce subprogramul își termină execuția: **P** ⇔ **SP**.

Parametrii de ieșire și parametrii de intrare-ieșire prin care subprogramul transmite rezultatele modulului apelant se mai numesc și **valori returnate** de către subprogram. La apelarea subprogramelor, parametrii de intrare pot fi și constante sau expresii, iar parametrii de ieșire și parametrii de intrare-ieșire pot fi numai variabile de memorie.

1.1.3. Elementele subprogramului

În limbajul C++ există trei elemente implicate în utilizarea unui subprogram:

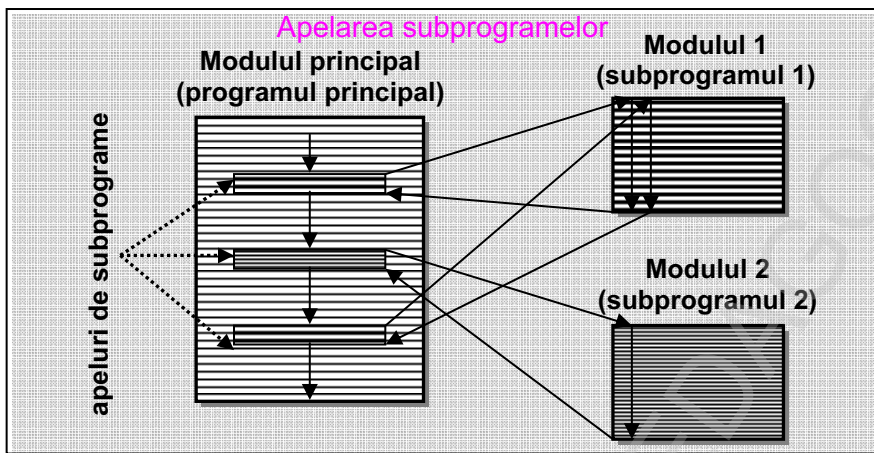
- **definiția subprogramului** – conține numele subprogramului, tipul argumentelor și al valorilor returnate și specifică ceea ce trebuie să realizeze subprogramul;
- **prototipul subprogramului** – comunică compilatorului informații despre subprogram (modul în care se poate apela subprogramul);
- **apelul subprogramului** – execută subprogramul.



Subprogramul se poate identifica printr-un **nume** care este folosit atât pentru **definiția subprogramului**, cât și pentru **prototip** și **activarea** lui (apelarea lui).

Apelarea subprogramului în cadrul unui bloc înseamnă **activarea subprogramului**, adică lansarea lui în execuție. Subprogramul poate fi apelat ori de câte ori este nevoie (nu există restricții pentru numărul de apeluri).

Modulul apelant se execută secvențial (instrucțiune cu instrucțiune). La apelarea subprogramului, este părăsit blocul modulului apelant și se trece la executarea instrucțiunilor din subprogram. După ce se termină executarea acestor instrucțiuni, se revine la blocul apelant și se continuă execuția cu instrucțiunea care urmează apelului.



1.1.4. Clasificarea subprogramelor

Pentru clasificarea subprogramelor se pot folosi două criterii:

- **modalitatea de apel**, determinat de modul de returnare a valorilor rezultate;
- **autorul**.

1.1.4.1. Clasificarea în funcție de modalitatea de apel

Subprogramele se împart în:

- **subprograme apelate ca instrucțiuni procedurale**;
- **subprograme apelate ca operanzi**.

Deoarece, în limbajul C++, toate subprogramele, indiferent de modul în care sunt apelate, se numesc **funcții**, pentru a identifica cele două tipuri de subprograme, le vom denumi ca funcții procedurale și funcții operand.

Funcții procedurale

Funcția procedurală este subprogramul care returnează una, mai multe sau nici o valoare. Valorile se returnează prin **intermediul parametrilor**.

Modalitatea de apel. Subprogramul se apelează printr-o **instrucțiune procedurală** care are următoarea sintaxă (o instrucțiune expresie care are un singur operand – apelul subprogramului):

nume_subprogram (listă_parametri);

optional

Observații:

1. În listă, parametrii sunt separați prin virgulă.
2. Parametrii pot fi nume de variabile de memorie, expresii sau valori constante. Ei se mai numesc și **argumentele** funcției.

Exemple de apeluri de funcții procedurale implementate în limbajul C++:

`clrscr();`

Apelul unei funcții procedurale fără parametri (**CLear SCREEN**) care șterge informațiile afișate pe ecranul calculatorului.

randomize () ;

Apelul unei funcții procedurale fără parametri care inițializează generatorul de numere aleatoare.

swab (s1, s2, n) ;

Apelul unei funcții procedurale cu trei parametri: copiază **n** caractere (**n** fiind un număr par), din șirul de caractere **s1**, la începutul șirului de caractere **s2**, inversând caracterele adiacente. Parametrul **s2** este un parametru de intrare-ieșire, iar parametrii **s1** și **n** sunt parametri de intrare.

gotoxy (x, y) ;

Apelul unei funcții procedurale cu doi parametri: în modul de lucru text, mută cursorul în fereastra de text curentă, în poziția precizată prin coordonatele **x** și **y**. Parametrii **x** și **y** sunt parametri de intrare.

Funcții operanzi

Funcția operand este un subprogram care returnează un rezultat prin chiar **numele său**, și eventual și alte rezultate, prin intermediul parametrilor.

Modalitatea de apel. Subprogramul se activează în interiorul unei expresii unde este folosit ca operand. Expresia poate să apară fie în membrul drept al unei instrucțiuni de atribuire, fie în cadrul unei instrucțiuni de control, fie în lista de parametri ai unei alte funcții (funcție operand sau instrucțiune procedurală). De exemplu:

nume_expresie = nume_variabilă +|-|*|/ **nume_funcție (listă_parametri);**

opțional

Observații:

1. La fel ca și la subprogramele apelate ca instrucțiuni procedurale, parametrii din listă sunt separați prin virgulă și pot fi nume de variabile de memorie, expresii sau valori constante.
2. În cazul funcției operand care returnează un singur rezultat prin chiar numele ei, parametrii din lista de parametri sunt de obicei numai parametri de intrare.
3. Funcția operand poate fi apelată și ca o funcție procedurală. În acest caz se pierde valoarea returnată prin numele ei.

Exemple de apeluri de funcții operand implementate în limbajul C++:

x=3.5; e=5+floor(x) ;

La calculul expresiei care se atribuie variabilei **e** se activează funcția **floor()** prin care se determină cel mai mare întreg mai mic decât valoarea parametrului. Funcția are un singur parametru – **x**, care este parametru de intrare și are valoarea 3.5. Rezultatul (data de ieșire) este furnizat prin numele funcției și are valoarea 3. Așadar, variabilei de memorie **e** i se va atribui valoarea: 8 (5+3).

for (i=0; i<=sqrt(n) ; i++) ;

La calculul expresiei ce se atribuie valorii finale a contorului structurii repetitive **for**, se activează funcția **sqrt(n)** care furnizează radicalul de ordinul 2 din valoarea parametrului. Funcția are un singur parametru – **n**, care este parametru de intrare.

```
x=sqrt(pow(3,2)+ pow(4,2));
```

La calculul expresiei care se atribuie variabilei **x** se activează de două ori funcția **pow()**: o dată pentru a calcula 3 la puterea 2, returnând valoarea 9, și o dată pentru a calcula 4 la puterea 2, returnând valoarea 16. Funcția **pow()** are doi parametri de intrare: primul este baza, iar al doilea este exponentul. Rezultatul este furnizat prin numele funcției. Rezultatul obținut prin evaluarea expresiei $9+16 = 25$ va fi parametru de intrare pentru funcția **sqrt()** care extrage radicalul de ordinul 2 din valoarea lui. Rezultatul funcției **sqrt()** – data de ieșire – este furnizat prin numele funcției și are valoarea 5. El este atribuit variabilei de memorie **x**. Așadar variabila de memorie **x** va avea valoarea 5.

1.1.4.2. Clasificarea în funcție de autor

Subprogramele se împart în:

- **subprograme standard** sau **subprograme de sistem**;
- **subprograme nestandard** sau **subprograme utilizator**.

Subprograme de sistem

Sunt subprograme predefinite de autorii limbajului de programare, care sunt furnizate împreună cu limbajul de programare. Ele se găsesc grupate, după funcțiile pe care le realizează, în **bibliotecile limbajului de programare**. Aceste subprograme rezolvă probleme generale ale utilizatorului, ca de exemplu:

- probleme matematice: calculul funcțiilor trigonometrice (*sin()*, *cos()* etc.), calculul unor funcții matematice (radicalul – *sqrt()*, exponențialul – *exp()*, logaritmul – *log()*, puterea – *pow()*), calculul părții întregi și fracționare dintr-un număr real (*modf()*, *fmod()*) etc.
- operații cu fișiere: deschiderea unui fișier – *fopen()*, închiderea unui fișier – *fclose()* etc.

Înainte de apelarea unui astfel de subprogram, trebuie făcut cunoscut compilatorului prototipul subprogramului, prin instrucțiunea pentru preprocesor:

```
#include <nume_fisier_antet.h>;
```

Așadar, lucrul cu un subprogram de sistem presupune două operații:

- **includerea fișierului care conține prototipul subprogramului în fișierul sursă al programului**,
- **apelarea subprogramului**.

Subprograme utilizator

Sunt subprograme create de programator pentru a rezolva unele sarcini (cerințe) specifice aplicației sale. Astfel, în exemplul de program prin care se verifică dacă trei numere pot forma o progresie aritmetică, pentru ordonarea celor trei numere **a**, **b** și **c**, programatorul poate construi un subprogram care să execute secvența de instrucțiuni prin care se realizează interschimbarea a două valori, secvență care se repetă de trei ori în cadrul programului.

Aceste subprograme trebuie declarate sau definite de programator înainte de apelul lor din funcția rădăcină sau dintr-un alt subprogram.

Lucrul cu un subprogram utilizator presupune două operații:

- **definirea subprogramului** și, dacă este cazul, **precizarea prototipului**.
- **apelarea subprogramului**.

1.1.5. Reguli pentru construirea subprogramelor C++

1.1.5.1. Definiția subprogramului

Definiția unui subprogram este formată din antetul și corpul subprogramului:

```
<antetul subprogramului>
{
    <declarații proprii subprogramului>
    <instrucțiuni>
    [return <expresie>;]
}
```

- Antetul subprogramului.** Este o linie de recunoaștere a subprogramului, în care i se atribuie un nume. El specifică începutul subprogramului.
- Corpul subprogramului.** La fel ca orice bloc C++, este încapsulat într-o instrucțiune compusă, delimitată de caracterele `{...}` și este format din două părți:
 - **Partea declarativă.** Conține definiții de elemente folosite numai în interiorul subprogramului: tipuri de date, constante și variabile de memorie. Nu se pot defini și alte subprograme (nu este valabilă tehnica de imbricare a subprogramelor existentă în alte limbaje de programare).
 - **Partea executabilă.** Conține instrucțiunile prin care sunt descrise acțiunile realizate de subprogram.

Antetul subprogramului

Subprogramul trebuie să aibă un antet prin care se precizează interfața dintre programul apelant și subprogram. El conține trei categorii de informații:

- **Tipul rezultatului.** Pentru funcțiile operand se precizează tipul rezultatului furnizat de subprogram prin chiar numele său. Pentru funcțiile procedurale, tipul rezultatului este **void** (nu întoarce nici un rezultat prin numele său; rezultatele vor fi întoarse prin parametrii subprogramului). Dacă nu se precizează tipul rezultatului, compilatorul va considera că acesta este implicit de tip **int**.
- **Numele subprogramului.** Este un identificator unic, care se atribuie subprogramului. Numele trebuie să respecte aceleași reguli ca orice identificator C++.
- **Parametrii folosiți pentru comunicare.** Pentru fiecare parametru se precizează numele și tipul.

Antetul unui subprogram este de forma:

```
tip_rezultat  nume_subprogram (listă_parametri)
```

Lista de parametri este de forma:

```
tip1 p1, tip2 p2, tip3 p3, ... tip_n p_n
```

tipul parametrului

identificatorul parametrului

separator

Exemplul 1:

```
float alfa(int a, int b, float c)
```

Acesta este antetul unei funcții operand care furnizează un rezultat de tip **float**. Numele funcției este *alfa*, iar parametrii folosiți pentru comunicare sunt *a* și *b* de tip **int** și *c* de tip **float**.

Exemplul 2:

```
void beta(int a, float b, float c, char d)
```

Acesta este antetul unei funcții procedurale. Numele funcției este *beta*, iar parametrii folosiți pentru comunicare sunt: *a* de tip **int**, *b* și *c* de tip **float** și *d* de tip **char**.

Exemplul 3:

```
void gama()
```

Acesta este antetul unei funcții procedurale. Numele funcției este *gama* și nu folosește parametri pentru comunicare.

Observații:

1. Separarea parametrilor în listă se face prin caracterul virgulă (.). Dacă există mai mulți parametri de același tip, ei **nu pot pot fi grupați ca la declararea tipului variabilelor de memorie**. Pentru fiecare parametru trebuie precizat tipul.
2. Tipul parametrilor poate fi:
 - orice tip standard al sistemului folosit pentru date elementare – întreg (**int**, **unsigned**, **long**), real (**double**, **float**, **long double**) sau caracter (**char** sau **unsigned char**) –, tipul pointer sau orice tip de structură de date (vector, matrice, șir de caractere sau înregistrare);
 - orice tip definit de utilizator înainte de a defini subprogramul.
3. Pentru rezultatul funcției nu se poate folosi tipul tablou de memorie.

Exemplu:

```
float a[10] tablou(int v, unsigned n)
```

Acest antet de subprogram va produce **eroare** deoarece tipul funcției este tablou de memorie.

4. Numele subprogramului poate fi folosit în trei locuri distincte:

- în **prototipul subprogramului**, unde are un rol declarativ;
- în **antetul subprogramului**, unde are un rol de definiție, dar și declarativ;
- în **apelul subprogramului**, unde are rol de activare.

Corpul subprogramului

Corpul subprogramului este un **bloc** care conține atât instrucțiuni declarative, cât și instrucțiuni imperative. Variabilele de memorie declarate în corpul subprogramului se numesc **variabile locale**. În cazul unei funcții operand, ultima instrucțiune din corpul subprogramului trebuie să fie instrucțiunea **return**, care are sintaxa:

```
return <expresie>;
```

Valoarea obținută prin evaluarea expresiei <expresie> va fi atribuită funcției operand (va fi valoarea returnată prin numele funcției). Rezultatul expresiei trebuie să fie de același tip cu tipul funcției sau de un tip care poate fi convertit implicit în tipul funcției.

Atenție

Când compilatorul C++ întâlnește într-un subprogram instrucțiunea **return**, termină execuția subprogramului și redă controlul modului apelant. Prin urmare, dacă veți scrie în subprogram, după instrucțiunea **return**, alte instrucțiuni, ele nu vor fi executate.

Exemplul 1:

```
int x,y; float z,w;
w = alfa(x,y,z);
```

Exemplul 2:

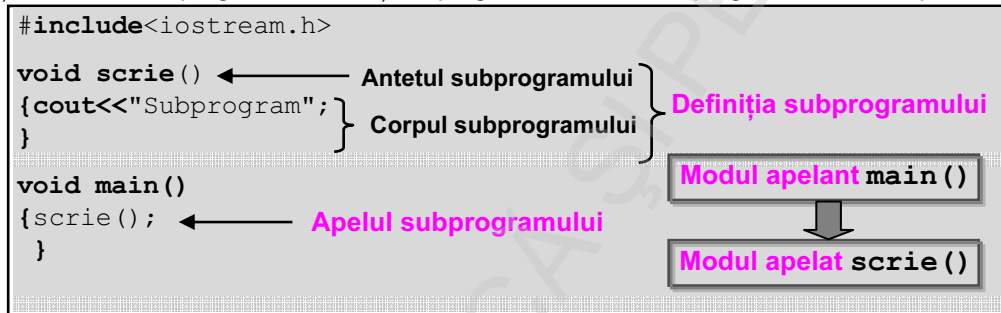
```
int x; float y,z; char w;
beta(x,y,z,w);
```

Exemplul 3:

```
gama();
```

Atenție

Orice subprogram trebuie **declarat** și **definit**. Declararea unui subprogram este necesară pentru ca el să fie cunoscut de subprogramele care îl apelează. Declararea lui poate fi făcută fie prin prototip, fie prin definiția lui (antetul împreună cu corpul subprogramului). Pentru a declara subprogramul, fie scrieți prototipul înaintea subprogramele care îl apelează, putând scrie apoi definiția lui oriunde în program, fie definiți subprogramul înaintea subprogramului care îl apelează.



Așadar:

- **Prototipul subprogramului** declară subprogramul.
- **Apelul subprogramului** execută subprogramul.
- **Antetul subprogramului** specifică numele subprogramului și tipul argumentelor și al valorilor returnate, iar corpul subprogramului îl definește, adică specifică ceea ce trebuie să realizeze subprogramul.

1.1.5.4. Parametrii de comunicare

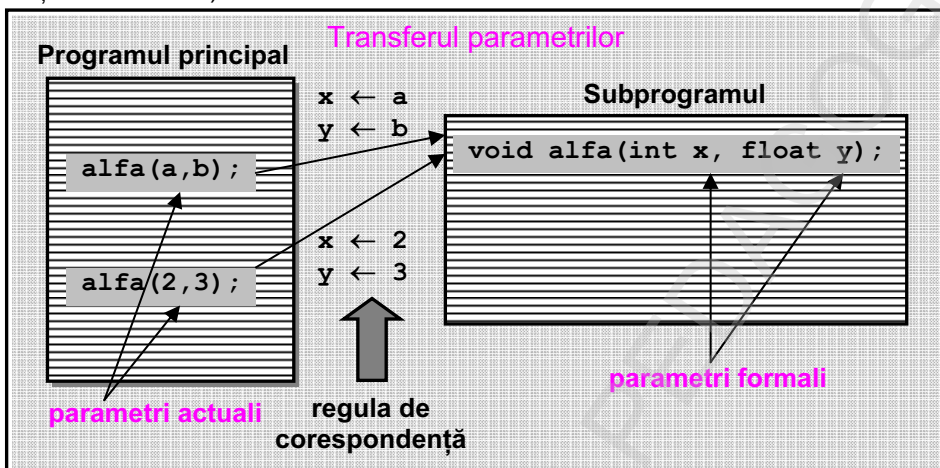
Dacă pentru comunicarea între subprogram și blocul apelant se folosesc parametri, aceștia vor fi scriși după numele subprogramului, între paranteze rotunde, astfel:

- În **antet**, pentru fiecare parametru se precizează denumirea simbolică folosită în interiorul subprogramului. Acești parametri se numesc **parametri formali**. În **prototip**, pentru fiecare parametru se precizează tipul de dată la care se referă, în aceeași ordine în care au fost scriși la definirea lor în antet.
- La **activarea** subprogramului, parametrilor de comunicare li se vor atribui valori concrete cu care se va executa subprogramul la acel apel. Aceste valori vor fi comunicate la apelul subprogramului, după numele subprogramului, între paranteze rotunde, în aceeași ordine în care au fost scriși la definirea lor în antet. Comunicarea trebuie să respecte aceeași succesiune, tipuri de date și număr de parametri ca și în lista de parametri formali, deoarece atribuirea valorilor se face respectând **regula de corespondență**. Acești parametri se numesc **parametri actuali**.

Observații.

1. Numele parametrilor actuali pot fi diferite de numele parametrilor formali.

2. Tipul parametrilor formali poate să fie diferit de tipul parametrilor actuali, numai când parametri efectivi pot fi **converțiți implicit** în tipul parametrilor formali (la fel ca în cazul operației de atribuire).



Studiu de caz

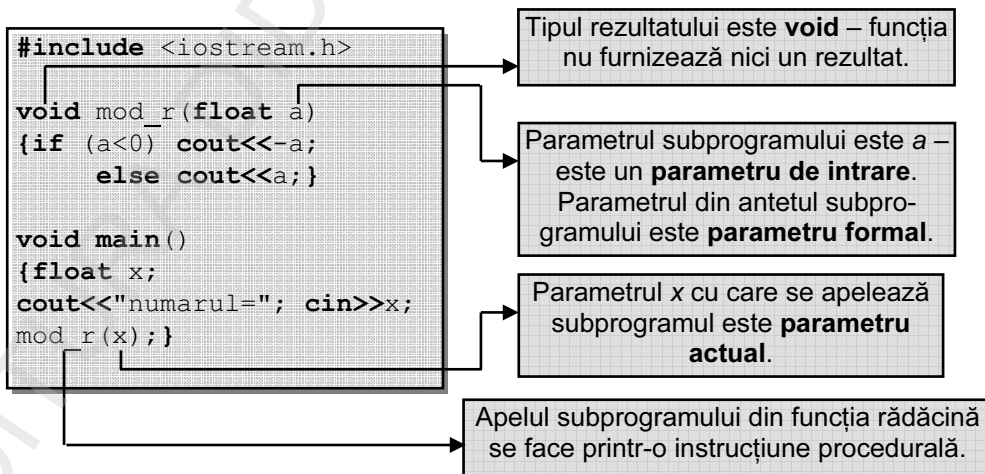
Scop: exemplificarea modului în care poate fi construit un subprogram C++.

Enunțul problemei: Să se construiască un subprogram care să calculeze valoarea absolută a unui număr real. Numele subprogramului este `mod_r`.

Acest subprogram va fi construit în două variante: ca funcție procedurală și ca funcție operand. În ambele cazuri modulul apelant va fi funcția rădăcină, iar modulul apelat va fi subprogramul `mod_r`.

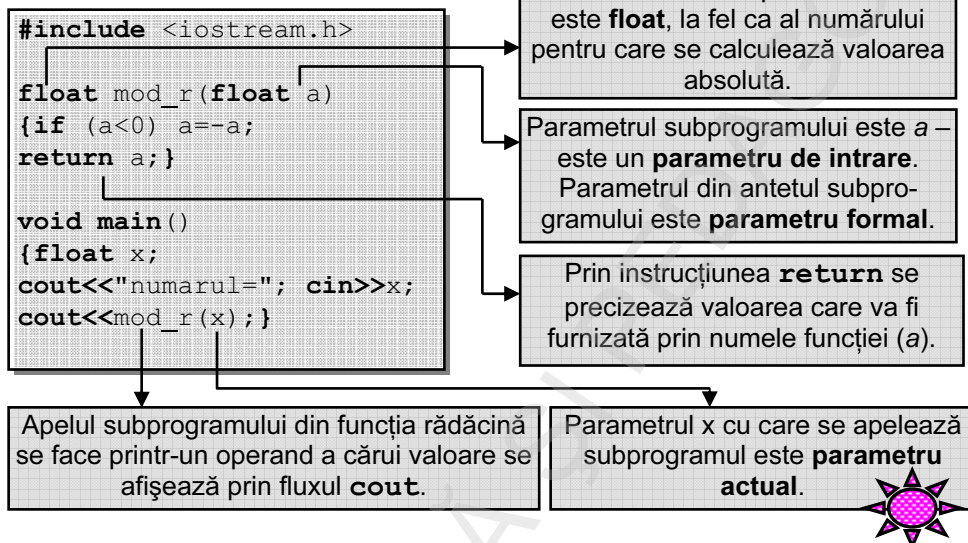
Varianta 1:

În cazul funcției procedurale, subprogramul va afișa valoarea modulului numărului și nu va furniza niciun rezultat funcției rădăcină care îl apelează. El va primi valoarea numărului de la funcția rădăcină prin intermediul parametrului.



Varianta 2:

În cazul funcției operand, subprogramul va returna funcției rădăcină, prin numele său, valoarea absolută a numărului. El va primi valoarea numărului de la funcția rădăcină prin intermediul parametrului.

**Temă**

1. Scrieți un program prin care să calculați aria unui triunghi. Valorile pentru laturile triunghiului se introduc de la tastatură în funcția rădăcină, iar aria se calculează într-un subprogram. Veți construi subprogramul în două variante:

- a) valoarea ariei se afișează în funcția rădăcină;
- b) valoarea ariei se afișează în subprogram.

Executați programele instrucțiune cu instrucțiune, folosind tasta **F7**.

2. Scrieți un subprogram care să returneze numărul de cifre ale unui număr natural, transmis ca parametru.

1.1.5.5. Utilizarea stivei de către subprograme

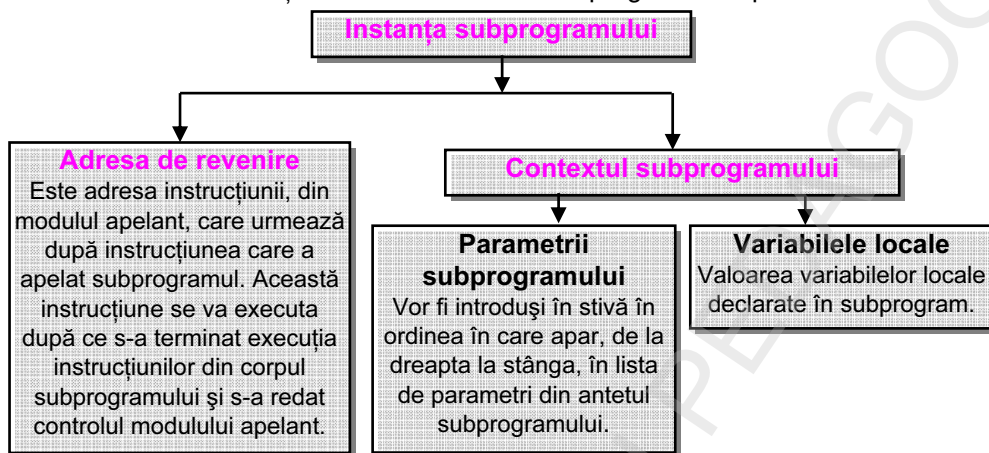
În memoria internă, fiecărui subprogram *i* se alocă o zonă de memorie în care este încărcat codul executabil. Ați văzut că, la apelarea unui subprogram, compilatorul îi predă controlul, adică încep să se execute instrucțiunile subprogramului, până la întâlnirea unei instrucțiuni **return** sau până la sfârșitul blocului care formează corpul subprogramului, după care compilatorul redă controlul modulului apelant, adică va continua execuția cu instrucțiunea care urmează, în modulul apelant, imediat după instrucțiunea care a apelat subprogramul. Acest mecanism de transfer al controlului se poate realiza deoarece, într-o zonă de memorie internă numită **stiva sistemului (stack)**, se păstrează temporar informații despre subprogramul apelant. Aceste informații sunt introduse în stivă atunci când este apelat subprogramul. Ele formează **instanța subprogramului**.

Etapele executate la apelarea subprogramului sunt:

1. Se întrerupe execuția modulului apelant.
2. Se pregătește stiva sistemului, astfel:

- se introduce adresa de revenire în modulul apelant;
- se introduc valorile parametrilor cu care a fost apelat subprogramul;
- se rezervă spațiu pentru variabilele locale declarate în subprogram.

3. Se lansează în execuție codul executabil al subprogramului apelat.



Etapele executate la terminarea subprogramului sunt:

1. Se eliberează din stivă spațiul ocupat de variabilele locale și de parametri.
2. Se extrage din stivă adresa de revenire în modulul apelant.
3. Se continuă execuția cu instrucțiunea de la adresa extrasă din stivă.

Astfel, pentru exemplele anterioare de declarații de subprograme, la apelarea lor, în stivă se vor introduce următoarele informații:

alfa (x, y, z)			beta (x, y, z, w)		gama ()	
variabile locale alfa			variabile locale beta		variabile locale gama	
x			x			
y			y			
z			z			
adresa de revenire			w			
			adresa de revenire		adresa de revenire	

Așadar, în timpul execuției subprogramului, în stivă sunt păstrate datele cu care el lucrează: variabilele locale și parametrii cu care a fost apelat. Instrucțiunile subprogramului pot modifica aceste date. Modificările se execută asupra valorilor memorate în stivă. Când se termină execuția subprogramului, trebuie să se reia execuția modulului apelant cu instrucțiunea de adresă de revenire. Pentru a se ajunge în stivă la adresa de revenire, spațiul ocupat de parametri și de variabilele locale este eliberat și se pierd valorile lor.

Studiu de caz

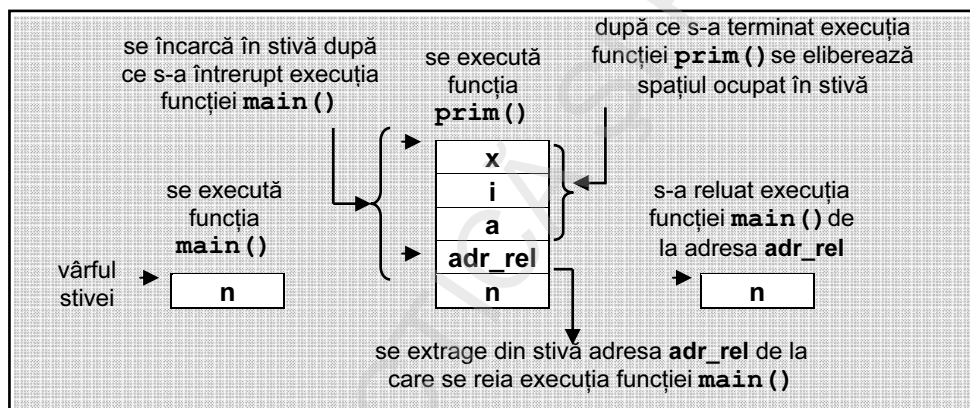
Scop: exemplificarea modulului în care este folosită stiva sistemului la apelarea unui subprogram.

Enunțul problemei: Să se verifice dacă un număr natural n , citit de la tastatură, este număr prim. Pentru testarea numărului se va folosi un subprogram.

Funcția **prim(a)** furnizează, prin numele său, o valoare întreagă ce poate fi interpretată ca o valoare logică: 0 – false sau 1 – true. În variabila locală **x** se calculează valoarea funcției **prim** (1 sau 0, în funcție de numărul **a** – dacă este sau nu este număr prim).

```
#include <iostream.h>
#include <math.h>
int prim (int a) //parametrul formal a
{int i,x=1; //variabilele locale în funcția prim()
  if(a%2==0&&a!=2) return 0;
    else {for(i=3;i<=sqrt(a)&&x;i++,i++)
          if(a%i==0) x=0;
    return x;}}
void main()
{int n; cout<<"n = "; cin>>n; //n=variabila locală în funcția main()
  if (prim(n)) cout<<"Este numar prim"; //parametrul actual n
    else cout<<"Nu este numar prim";}
```

Conținutul stivei sistemului va fi:



Temă



1. Scrieți un subprogram în care calculați cel mai mare divizor comun a două numere naturale (**a** și **b**). Folosiți acest subprogram pentru a calcula cel mai mare divizor comun a **n** numere introduse de la tastatură. Arătați care este conținutul stivei în timpul execuției programului.
2. Folosiți subprogramul care testează dacă un număr natural este număr prim, pentru a rezolva următoarea problemă: se introduc de la tastatură **n** numere naturale; să se afișeze câte numere sunt prime.
3. Folosiți subprogramul pentru calculul celui mai mare divizor comun a două numere, pentru a rezolva următoarea problemă: se introduc de la tastatură două numere, **s** și **m**; să se afișeze toate perechile de numere care au suma **s** și cel mai mic multiplu comun **m**.



1.1.6. Transferul de parametri între subprograme

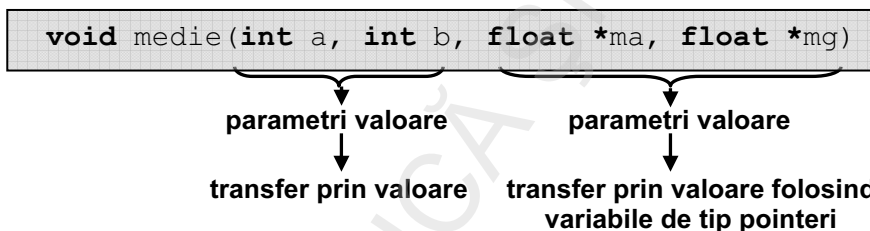
Schimbul de date între modulul apelant și subprogram se face prin intermediul parametrilor de comunicație.

Transferul de parametri este o tehnică folosită pentru schimbul de date între module.

Există următoarele **metode de transfer**:

1. Transfer prin valoare

- Modulul apelant transmite prin parametru, către subprogram, date de intrare. În momentul apelării subprogramului, o copie a valorii parametrului este încărcată în stivă. El este văzut în subprogram ca o variabilă locală, care este inițializată cu valoarea transmisă de modulul apelant prin parametrul actual din apel. Valoarea acestei variabile se poate modifica în subprogram, dar această modificare nu se va reflecta și în modulul apelant, deoarece modificarea se face în stivă, și, la terminarea execuției subprogramului, zona din stivă în care este memorat parametrul este eliberată.
- Parametrul prin intermediul căruia se face transferul prin valoare se numește **parametru valoare**.
- Acest transfer se folosește în general numai pentru **parametrii de intrare**. În cazul în care parametrii transmiși prin valoare sunt parametri de ieșire sau de intrare-ieșire, pentru a putea transmite rezultatul obținut în subprogram, către modulul apelant, se pot folosi variabile de tip **pointeri** (sunt prezentați în Anexă).
- **Exemplu de antet de subprogram** pentru un astfel de transfer (subprogramul furnizează, prin parametrii **ma** și **mg**, media aritmetică, și respectiv media geometrică, a două numere transmise subprogramului prin parametrii **a** și **b**).



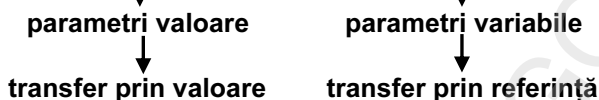
Apelarea acestui subprogram se va face prin: `medie(x, y, &m1, &m2);` Parametrilor **a** și **b** li se transmit, din modulul apelant, valorile variabilelor **x** și respectiv **y**, iar parametrilor de tip pointer, **ma** și **mb**, valoarea adreselor variabilelor **m1** și respectiv **m2**.

2. Transfer prin referință

- În momentul apelării subprogramului, în stivă este încărcată adresa de memorie la care se găsește valoarea parametrului. Subprogramul va lucra direct în zona de memorie în care se găsește data. Atât modulul apelant cât și subprogramul lucrează asupra aceleiași date, și orice modificare a valorii acestui parametru făcută în subprogram se va reflecta și în modulul apelant. La terminarea execuției subprogramului, este eliberată din stivă zona în care este memorată adresa parametrului.
- Parametrul prin intermediul căruia se face transferul prin referință se numește **parametru variabilă**.
- Acest transfer se recomandă pentru **parametrii de intrare-ieșire** sau **parametrii de ieșire**. Modulul apelant transmite, prin acești parametri, date de intrare-ieșire către subprogram, subprogramul preia data, o prelucrează și o returnează modulului apelant. Acest parametru mai poate fi și un rezultat (dată de ieșire) obținut în urma prelucrărilor din subprogram, care este returnat apoi modulului apelant.
- Distincția dintre un parametru valoare și un parametru variabilă (definirea tipului de transfer) se face în **lista de parametri formali din antetul subprogramului** în care **parametrii variabilă** sunt precedați de operatorul adresă de memorie **&**.

→ **Exemplu de antet de subprogram** pentru un astfel de transfer (pentru un subprogram care rezolvă aceeași problemă ca și în exemplul precedent):

```
void medie (int a, int b, float &ma, float &mb)
```



Apelarea acestui subprogram se va face prin: `medie (x, y, m1, m2) ;`

Din modulul apelant se transmit: parametrilor **a** și **b**, care sunt parametri valoare – valorile variabilelor **x** și respectiv **y** –, iar parametrilor **ma** și **mb**, care sunt de tip parametri variabilă – adresele variabilelor **m1** și respectiv **m2**.

Observație:

Pentru **transmiterea unor rezultate** din subprogram către modulul apelant (parametru de ieșire sau de intrare-ieșire) se folosește fie **transferul prin referință**, fie **transferul prin valoare**, folosind variabile de tip **pointeri**.

Observații:

- Parametrii actuali** corespunzători **parametrilor valoare** pot fi exprimați prin:
 - valoare (constantă);
 - expresie;
 - variabilă de memorie;
 - adresă a unei variabile de memorie (este obligatorie, în cazul în care parametrii formali sunt de tip pointer).
- Parametrii formali** corespunzători **parametrilor valoare** pot fi inițializați în antetul subprogramului. La apelul subprogramului, parametrilor formali li se atribuie valoarea parametrilor actuali. Dacă lipsește un parametru actual, parametrul formal va fi inițializat cu valoarea din listă:

```
#include<iostream.h>
int test(int a=10, int b=20) {return a+b;}
void main()
{cout<<test(30,40)<<endl;      //afișează 70
cout<<test(30)<<endl;          //afișează 50
cout<<test();                  //afișează 30}
```

- Parametrii actuali** corespunzători **parametrilor variabilă** pot fi exprimați **numai prin variabile de memorie**.

Studiu de caz

Scop: exemplificarea modului în care pot fi transmiși parametrii între subprograme.

Enunțul problemei: Să se construiască un subprogram care să realizeze *inter-schimbarea valorilor a două variabile de memorie întregi*.

Numele subprogramului este **schimb**. Modulul apelant va fi funcția rădăcină, iar modulul apelat va fi subprogramul **schimb**. Din funcția rădăcină se vor transfera subprogramului parametri **x** și **y**, care reprezintă variabilele a căror valoare se interschimbă. Acest subprogram va fi construit în trei variante, în funcție de modul în care sunt transferați parametrii:

Varianta 1

Transferul parametrilor se face prin valoare.

```
#include<iostream.h>
int schimb(int x, int y)
{int z;
z=x; x=y; y=z;}
void main()
int a,b;
cout<<"a= "; cin>>a;
cout<<"b= "; cin>>b;
schimb(a,b);
cout<<a<<" "<<b;}
```

Varianta 2

Transferul parametrilor se face prin valoare, folosind variabile de tip pointer.

```
#include<iostream.h>
int schimb(int *x, int *y)
{int z;
z=*x; *x=*y; *y=z;}
void main()
int a,b;
cout<<"a= "; cin>>a;
cout<<"b= "; cin>>b;
schimb(&a,&b);
cout<<a<<" "<<b;}
```

Varianta 3

Transferul parametrilor se face prin referință.

```
#include<iostream.h>
int schimb(int &x, int &y)
{int z;
z=x; x=y; y=z;}
void main()
int a,b;
cout<<"a= "; cin>>a;
cout<<"b= "; cin>>b;
schimb(a,b);
cout<<a<<" "<<b;}
```

Temă



Comparați cele trei variante ale subprogramului *schimb*. Executați fiecare variantă, pentru următoarele date de intrare: $a=10$ și $b=20$. Ce constatați? Explicați rezultatele obținute. Desenați diagrama stivei pentru fiecare variantă de transfer de parametri.



Temă



1. Scrieți un program în care să calculați media aritmetică (**ma**) și media geometrică (**mg**) a două numere întregi (**a** și **b**) introduse de la tastatură. În funcția rădăcină se citesc valorile pentru **a** și **b** și se afișează valorile celor două medii care se vor calcula în subprogramul **medie**. Veți implementa două variante pentru subprogram, pornind de la următoarele anteturi:
 - a. **void medie(int a, int b, float *ma, float *mg)**
 - b. **void medie(int a, int b, float &ma, float &mg)**
2. Scrieți un subprogram care returnează prima cifră și numărul de cifre ale unui număr natural **n** transmis ca parametru. De exemplu, pentru $n=608$, subprogramul returnează valorile 6 și 3.

(Bacalaureat – Sesiunea iunie - iulie 2004)

3. Realizați următoarele cerințe, utilizând limbajul C++:
 - a. Scrieți definiția unui subprogram **mindiv** care determină cel mai mic dintre divizorii mai mari decât 1 ai unui număr natural transmis prin intermediul parametrului **a** ($a>1$) și returnează acest divizor prin intermediul parametrului **b**.
 - b. Scrieți programul care citește două numere naturale **a** și **b** ($a<b$) și determină cel mai mare număr prim din intervalul închis $[a,b]$ cu ajutorul subprogramului definit la punctul a). Dacă nu există un astfel de număr, se va afișa mesajul **Nu exista**.

(Bacalaureat – Simulare 2006)

4. Se știe că este definită o funcție **nrap** care:

- primește prin intermediul parametrului **nr** un număr natural de cel mult 9 cifre și prin intermediul parametrului **cif** o cifră între 0 și 9;
- returnează numărul de apariții ale cifrei **cif** în scrierea numărului **nr** în baza 10. De exemplu, pentru numărul **2535** și cifra 5, se returnează 2, iar pentru numărul 2535 și cifra 7, se returnează 0.

a. Scrieți antetul subprogramului **nrap**.

b. Scrieți declarațiile de date și programul principal, în care se verifică dacă un număr natural **k** citit de la tastatură are toate cifrele distincte sau nu, folosind apeluri ale subprogramului **nrap**.

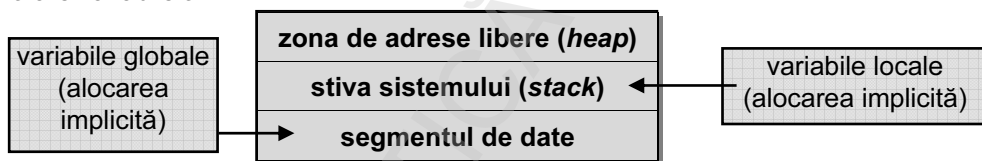
(Bacalaureat – Simulare 2004)

1.1.7. Clasificarea variabilelor de memorie

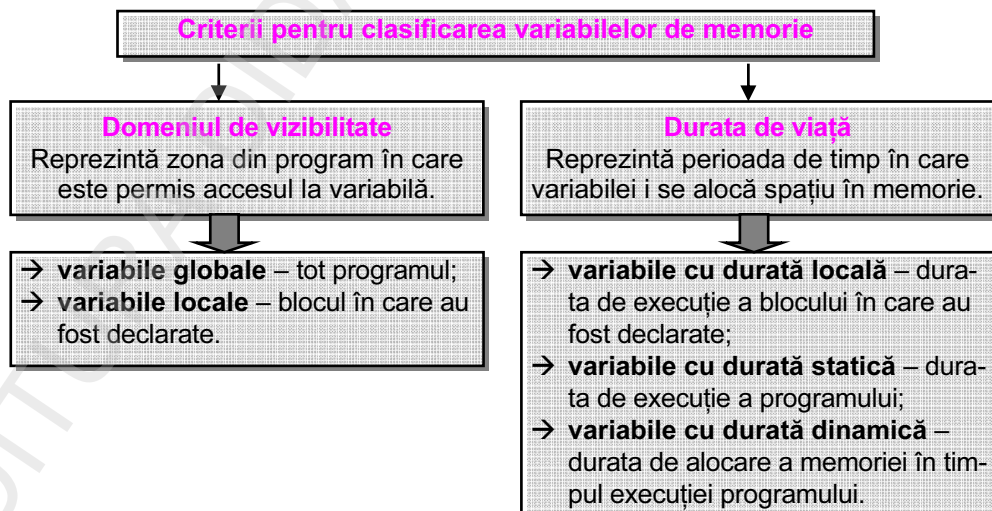
Unui subprogram aflat în execuție i se rezervă propriul spațiu de memorie în interiorul zonei de memorie rezervată programului principal. Subprogramele pot fi privite ca blocuri care conțin, pe lângă instrucțiuni, și alte obiecte, precizate prin:

- lista de parametri formali;
- instrucțiuni declarative din zona declarativă a subprogramului;
- instrucțiuni declarative într-un bloc din subprogram.

Există mai multe **zone de memorie** în care sistemul de operare poate alocă spațiu de memorare variabilelor:



Deoarece atât în modulul apelant cât și în subprogram sunt definiți mai mulți identificatori pentru aceste obiecte (variabile de memorie și constante) problema care se pune este: **care este domeniul de vizibilitate al identificatorilor, în funcție de locul în care sunt declarați**, și care este **durata de viață** a unei variabile de memorie? Ținând cont de aceste caracteristici ale variabilelor de memorie, ele pot fi clasificate după următoarele criterii:



1.1.7.1. Durata de viață a variabilelor de memorie

În funcție de durata de viață, variabilele de memorie se clasifică în:

1. Variabile cu durată locală

- Sunt variabile create în interiorul unui subprogram; compilatorul le creează și le distruge automat, atunci când începe execuția subprogramului, respectiv când se termină execuția lui.
- La fiecare nouă apelare a subprogramului, vor avea o valoare nedefinită (valoarea reziduală din zona de stivă care li se alocă). De aceea, ele trebuie întotdeauna inițializate.
- Sunt păstrate temporar în **stivă** (numai pe timpul execuției subprogramului).

2. Variabile cu durată statică

- Sunt create atunci când începe execuția subprogramului și durează pe tot timpul execuției subprogramului; ele corespund de regulă variabilelor globale.
- Sunt inițializate cu valoarea 0.
- Spațiul de memorie li se alocă la compilare, în **segmentul de date**.

3. Variabile cu durată dinamică

- Sunt create în timpul execuției programului și durează atât timp cât sunt necesare.
- Spațiul de memorie li se alocă în timpul execuției programului, în zona de memorie liberă (**heap**).

1.1.7.2. Domeniul de vizibilitate al identificatorilor

Domeniul de vizibilitate al identificatorilor este o caracteristică a oricărui identificator (nume de variabilă, nume de constantă, nume de funcție, nume de tip de dată) și reprezintă zona de program în care un identificator definit poate fi referit (este vizibil). De exemplu, dacă două variabile de memorie cu același nume au fost declarate în subprograme diferite, vor avea fiecare dintre ele ca domeniu de vizibilitate subprogramul în care au fost declarate. În funcție de domeniul de vizibilitate, variabilele de memorie se clasifică în:

1. Variabile locale

- Sunt variabile **definite în corpul unui subprogram**, în orice bloc al subprogramului (în orice instrucțiune compusă) și sunt variabile proprii subprogramului.
- Asupra lor pot fi executate operații (sunt **vizibile**) numai din interiorul blocului în care au fost declarate (blocul subprogramului sau blocul unei instrucțiuni compuse).
- Folosirea lor este utilă pentru a se elimina confuziile care apar atunci când se folosesc variabile cu același nume în două subprograme diferite.
- Variabilele definite în interiorul funcției rădăcină **main()** sunt tot variabile locale (sunt definite într-un bloc).
- **Durata lor de viață este locală** – numai pe perioada execuției blocului (subprogram sau instrucțiune compusă).
- La declarare, variabilele locale nu sunt inițializate cu o valoare. Ele păstrează valoarea atribuită anterior zonei de memorie alocate (valoare reziduală).
- **Parametrii formali** ai unui subprogram sunt **variabile locale**.
- Variabilelor locale li se alocă spațiu în stivă de fiecare dată când se apelează subprogramul. Când subprogramul își termină execuția, variabilele locale sunt eliminate din stivă și se eliberează spațiul ocupat de ele.

2. Variabile globale

- Sunt variabile **definite în afara oricărui subprogram**.

- Ele sunt **vizibile** în toate subprogramele care sunt declarate după definirea lor. Oricare dintre aceste subprograme poate folosi și modifica valoarea lor.
- Folosirea lor este utilă atunci când unele date se folosesc în comun de către modulele unui program care nu se apelează unele pe altele.
- **Durata lor de viață este statică** – pe toată perioada execuției programului (din momentul în care au fost declarate și până în momentul în care se termină execuția programului).
- Domeniul de vizibilitate al unei variabile globale poate fi controlat în funcție de locul în care o declarați, ținând cont de următoarea observație: atunci când declarați o variabilă globală, ea va putea fi folosită **de orice subprogram declarat după ea**, dar nu poate fi folosită de un subprogram declarat înaintea ei.
- La declarare, variabilele globale sunt inițializate cu valoarea 0.

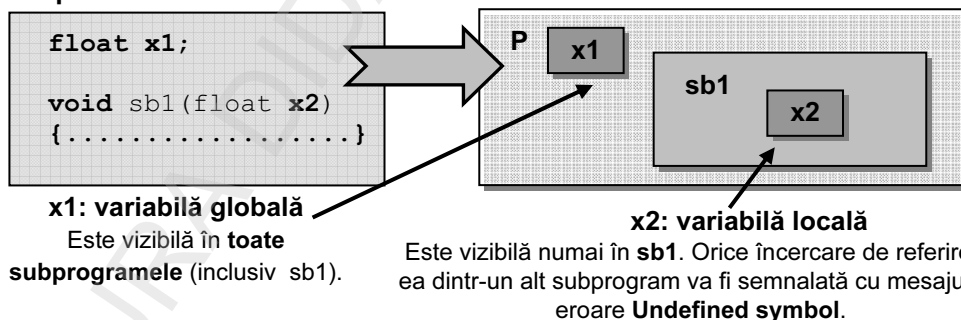
Reguli pentru vizibilitatea identificatorilor:

1. **În interiorul unui bloc, un identificador nu poate fi definit decât o singură dată.** Dacă veți folosi același nume pentru două obiecte diferite (variabilă, constantă sau tip de dată), apariția acestui nume în cadrul unei instrucțiuni va produce eroare la compilare.
2. **Un identificador nu poate fi folosit în exteriorul blocului în care a fost definit.**
3. **Un identificador poate fi declarat în blocuri diferite, de același tip sau de tipuri diferite.** În acest caz se rezervă câte o zonă de memorie pentru fiecare identificador declarat, de dimensiune corespunzătoare tipului declarat. Pentru un identificador folosit într-o instrucțiune, se stabilește tipul și zona de memorie alocată, căutându-se în cel mai interior bloc care conține atât instrucțiunea, cât și declarația identicatorului.

Conflictele de nume între variabile definite în domenii incluse unul în altul se rezolvă astfel: compilatorul ascunde temporar identicatorul exterior, care își pierde vizibilitatea:

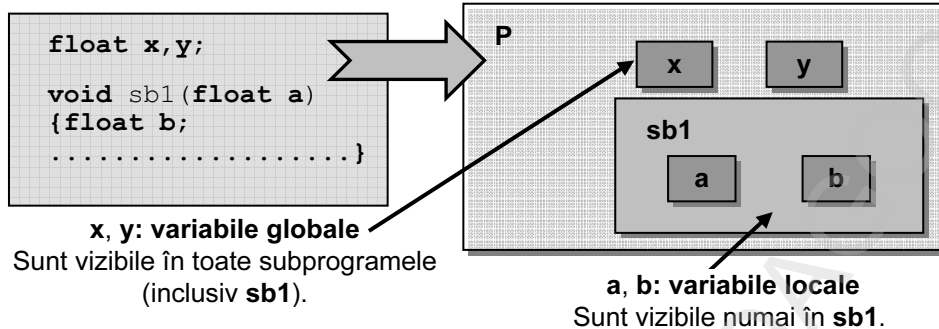
- Dacă există o variabilă globală și o variabilă locală cu aceleași nume, în subprogramul în care s-a definit variabila locală compilatorul va folosi întotdeauna variabila locală.
- Dacă există două variabile locale cu același nume, una definită pentru tot subprogramul, iar cealaltă într-un bloc din același subprogram, compilatorul va ascunde variabila locală definită pentru tot subprogramul, pe durata execuției blocului în care a mai fost definită.

Exemplul 1:



Instrucțiunea **x1=10;** poate să apară în orice subprogram, în schimb instrucțiunea **x2=10;** poate să apară numai în subprogramul **sb1** și afectează numai valoarea variabilei din acest subprogram.

Exemplul 2:

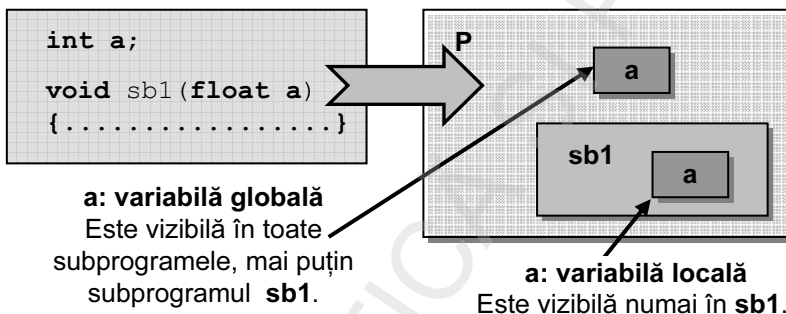


Instrucțiunea **cout<<x<<y<<a<<b;**

→ poate să apară în **sb1** (deoarece aici sunt vizibile toate variabilele);

→ nu poate să apară în nici un alt subprogram (deoarece nu sunt vizibile variabilele **a** și **b**).

Exemplul 3:

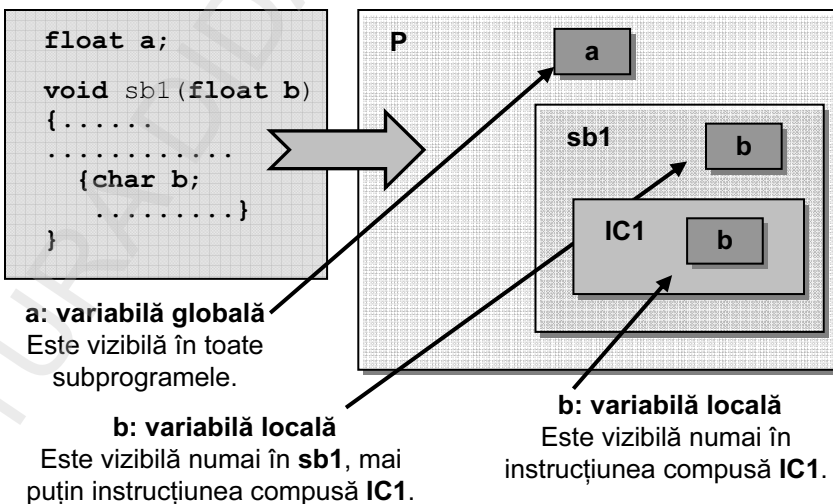


Instrucțiunea **a=10;**

→ dacă se găsește în orice subprogram diferit de **sb1**, afectează valoarea variabilei globale **a** (de tip **int**);

→ dacă se găsește în subprogramul **sb1** afectează valoarea variabilei locale **a** (de tip **float**).

Exemplul 4:



Instrucțiunea **b=10;**

- dacă se găsește în orice subprogram diferit de **sb1**, va fi considerat identificator necunoscut;
- dacă se găsește oriunde în subprogramul **sb1**, mai puțin în instrucțiunea compusă **IC1**, afectează valoarea variabilei locale **b** de tip **float**;
- dacă se găsește în instrucțiunea compusă **IC1**, afectează valoarea variabilei locale **b** de tip **char**.

Recomandări:

1. Valoarea unei **variabile globale** poate fi modificată din interiorul oricărui subprogram. Din această cauză, programatorul trebuie să fie foarte atent la aceste variabile. În plus, folosirea lor **degenerează coerența programării modulare**, deoarece creează dependențe între module, la proiectarea lor.
2. **Variabilele locale protejează integritatea programelor**, adică modificarea acestor variabile într-un subprogram nu afectează și alte subprograme. Ele asigură astfel **portabilitatea subprogramelor** (independența unui subprogram față de alte subprograme).
3. **Constantele globale** sunt avantajoase deoarece **permit modificarea unei valori constante în întreg programul**, adică, schimbând valoarea constantei în programul principal, actualizarea ei va fi văzută în toate subprogramele.

Este **recomandabilă folosirea parametrilor în locul variabilelor globale** deoarece:

- Prin variabilele globale datele sunt partajate între toate modulele. Orice modul poate modifica aceste date. Din această cauză, se creează dependențe nedorite între module.
- Parametrii permit o identificare clară a datelor partajate de anumite module, prin precizarea explicită a acestor date în lista de parametri din antetul subprogramului și prin lista de parametri din instrucțiunea cu care se apelează subprogramul.

Observații:

Transferul datelor (**comunicarea**) între subprogram și modulul apelant se poate face prin intermediul **parametrilor** sau al **variabilelor globale**.

1. Un subprogram, cu cât are **mai puțini parametri** (folosește mai multe variabile globale), cu atât este mai ușor de scris și apelat.
2. Un subprogram, cu cât are **mai mulți parametri** (folosește mai puține variabile globale), cu atât este mai flexibil, mai portabil (poate fi implementat ușor în programe diferite și nu influențează modulul apelant).

Atenție

Parametrii se deosebesc de variabilele globale. Ei sunt inițializați, în momentul apelului, cu valori primite din modulul apelant (valorile sunt parametrii actuali):

- în cazul **transferului prin valoare**, i se atribuie o valoare, o expresie sau conținutul unei variabile de memorie;
- în cazul **transferului prin referință**, subprogramul primește o adresă de memorie la care este stocată variabila primită ca parametru.

Studiu de caz

Scop: exemplificarea domeniului de vizibilitate și a duratei de viață a variabilelor de memorie.

Executați următorul program. Precizați tipul fiecărei variabile de memorie vizibile în fiecare dintre funcții. Notați valorile afișate pe ecran. Explicați rezultatele afișate.

```
#include<iostream.h>
int i; float x; char c;
void sb1(int a)    //se pot folosi variabilele a,b,c,i,x
{int i, b=10;
i=b; b+=a; a=i;
{int i;
for (i=1;i<=3;i++) cout<<i<<endl; }
cout<<i<<" "<<b<<endl; }
float a;
void sb2(float x) //se pot folosi variabilele a,b,c,i,x
{int b,c; a=x; b=c=x;
cout<<a<<" "<<b<<endl; }
void main()        //se pot folosi variabilele a,c,i,x
{int x=100; a=20.5; c='A';i=30;
sb1(20); sb2(a); cout<<a<<endl;
sb1(x); sb2(2.5); cout<<a<<endl;
sb1(c); sb2(i); cout<<a<<endl;}
```



1.1.8. Alegerea modului de implementare a subprogramului

Implementarea unui subprogram se poate face ca funcție operand sau ca funcție procedurală. Alegerea modului de implementare diferă în funcție de problema care trebuie rezolvată.

Studiu de caz

Scop: exemplificarea modului în care pot fi implementate subprogramele.

Enunțul problemei 1: Se consideră trei numere naturale, **a**, **b** și **c**. Să se verifice dacă pot forma o progresie aritmetică.

Pentru rezolvarea problemei, vom folosi subprogramul **schimb** care va fi implementat inițial ca funcție procedurală și apoi ca funcție operand. Va fi apelat de trei ori, pentru a ordona crescător cele trei numere.

Subprogramul implementat ca funcție procedurală va folosi numai parametrii de comunicare, pentru schimbul de date cu funcția rădăcină. Pentru a ordona crescător cele trei numere, subprogramul va fi apelat de trei ori, folosind o instrucțiune procedurală.

În antetul funcției procedurale **void schimba(int &x,int &y);** parametrii **x** și **y** sunt **parametri formali**, adică variabile de memorie folosite în cadrul subprogramului. Ei sunt **parametri de intrare-ieșire** deoarece sunt folosiți pentru a transmite date dinspre program către subprogram (ca date de intrare în subprogram), și invers, la terminarea execuției subprogramului, dinspre subprogram către program (ca date de ieșire din subprogram). În apelurile funcției procedurale (instrucțiunile procedurale) **schimba(a,b);**, și respectiv **schimba(b,c);**, parametrii **a** și **b** și respectiv **b** și **c** sunt **parametrii actuali**, adică valorile atribuite pentru datele declarate în funcție ca parametri de comunicare cu care se execută subprogramul la acel apel. Așadar, la apelul **schimba(a,b);** variabilei **x** i se va atribui valoarea variabilei **a**, iar variabilei **y** i se va atribui valoarea variabilei **b**. După executarea subprogramului, prin variabilele **x** și **y** se vor comunica rezultatele obținute în urma executării subprogramului, adică variabila **a** va avea valoarea variabilei **x** (în urma interschimbării, vechea valoare a variabilei **b**), iar variabila **b** va avea valoarea variabilei **y**.

(în urma interschimbării, vechea valoare a variabilei **a**). În schimb, la apelul `schimba(b, c)`; variabilei **x** i se va atribui valoarea **b**, iar variabilei **y** i se va atribui valoarea **c**.

Programul obținut prin folosirea **funcției procedurale** este:

```
#include <iostream.h>

void schimba(int &,int &); ← prototipul subprogramului

void main()
{int a,b,c;
cout<<"a= "; cin>>a;
cout<<"b= "; cin>>b;
cout<<"c= "; cin>>c;
if (a>b) schimba(a,b);
if (b>c) schimba(b,c);
if (a>b) schimba(a,b);
if (b==(a+c)/2.)
    cout<<a<<"", "<<b<<"", "<<c<<"sunt in progresie aritmetica";
else
    cout<<a<<"", "<<b<<"", "<<c<<"nu sunt in progresie aritmetica";}

void schimba(int &x,int &y) ← antetul subprogramului
{int z;
z=x; x=y; y=z;
} ← subprogramul

    parametri actuali
    apeluri de subprogram
    parametri formali
```

Observații rezultate din exemplul prezentat:

1. Citirea și scrierea datelor prelucrate de algoritm se execută în programul principal.
2. Comunicarea datelor între modulele apelante și modulele apelate se face numai prin intermediul perechilor **parametru formal – parametru actual**.
3. Parametrilor dintr-o pereche **parametru formal – parametru actual** li se poate atribui același nume. Obligatoriu este însă să se respecte același tip de dată în cadrul perechii și ordinea de scriere a parametrilor în cele două liste prin care se asigură regula de corespondență.

Temă



În exemplul prezentat, explicați modul în care se face transferul datelor între funcția rădăcină și subprogram, prin intermediul parametrilor de comunicare.

Subprogramul implementat ca funcție operand va furniza funcției rădăcină un rezultat prin numele subprogramului, iar celălalt printr-un parametru de intrare-ieșire. Pentru a ordona crescător cele trei numere, subprogramul va fi apelat de trei ori, ca operand în instrucțiuni de atribuire, prin care valoarea returnată de funcție se atribuie uneia dintre variabilele care se interschimbă, iar celelalte variabile i se transmite noua valoare prin intermediul unui parametru al subprogramului.

Programul obținut prin folosirea **funcției operand** este:

```
#include<iostream.h>
int schimba(int &,int);
void main()
{int a,b,c;
```

```
cout<<"a= "; cin>>a; cout<<"b= "; cin>>b; cout<<"c= "; cin>>c;
if (a>b) b=schimba(a,b);
if (b>c) c=schimba(b,c);
if (a>b) b=schimba(a,b);
if (b==(a+c)/2.)
    cout<<a<<"<<b<<"<<c<<"sunt in progresie aritmetica";
else cout<<a<<"<<b<<"<<c<<"nu sunt in progresie aritmetica";}
int schimba(int &x,int y)
{int z; z=x; x=y; y=z;
return y;}
```

În acest caz, **x** este un **parametru de intrare-ieșire**, deoarece este folosit pentru a transmite date, ca și în exemplul precedent, între funcția rădăcină și subprogram, iar **y** este un **parametru de intrare**, fiind folosit pentru a transmite date numai dinspre funcția rădăcină către subprogram (ca date de intrare în subprogram). În urma prelucrărilor din subprogram, rezultatul obținut în variabila de memorie **y** va fi transmis programului prin numele funcției. În apelurile funcției operand (expresiile din instrucțiunile de atribuire) **b=schimba(a,b)**; și respectiv **c=schimba(b,c)**; , parametri **a** și **b** și respectiv **b** și **c** sunt **parametrii actuali**, adică valorile atribuite pentru datele declarate în funcția operand ca parametri de comunicare cu care se execută subprogramul la acel apel. Așadar, la apelul **b=schimba(a,b)**; variabilei **x** i se va atribui valoarea variabilei **a**, iar variabilei **y** i se va atribui valoarea variabilei **b**. După executarea subprogramului, prin variabila **x** și prin numele funcției (căreia i se returnează valoarea variabilei **y**) se vor comunica rezultatele obținute în urma executării subprogramului, adică variabila **a** va avea valoarea variabilei **x** (în urma interschimbării, vechea valoare a variabilei **b**), iar variabila **b** va avea valoarea returnată de funcție prin numele ei (în urma interschimbării, vechea valoare a variabilei **a**). În schimb, la apelul **c=schimba(b,c)**; variabilei **x** i se va atribui valoarea **b**, iar variabilei **y** i se va atribui valoarea **c**.

Enunțul problemei 2: Să se rezolve calculul combinărilor definite astfel:

$$C(n,k) = n!/(k!(n-k)!)$$

Pentru a calcula valoarea combinărilor, trebuie evaluate trei expresii factorial: $p1=n!$, $p2=k!$ și $p3=(n-k)!$. Calcularea acestor expresii se face folosind aceeași metodă:

```
for(i=1;i<=n;i++) p1=p1*i;
for(i=1;i<=n-k;i++) p2=p2*i;
for(i=1;i<=k;i++) p3=p3*i;
```

Cei trei algoritmi de calcul se deosebesc numai prin valoarea finală a contorului **i**. Pentru implementarea lor se poate folosi un subprogram **fact** care se va executa de trei ori, de fiecare dată cu alte date de intrare (**n**, **k** și respectiv **n-k**).

Pentru calculul factorialului se poate folosi un subprogram, fie de tip funcție procedurală, fie de tip funcție operand, astfel:

Programul obținut prin folosirea funcției procedurale este:

```
#include<iostream.h>
void fact(int n,unsigned long &p)
{for(int i=1,p=1;i<=n;i++)p=p*i;}
void main()
```

```

{int n,k; unsigned long p1,p2,p3;
 cout<<"n= "; cin>>n; cout<<"k= "; cin>>k;
 fact(n,p1); fact(k,p2); fact(n-k,p3);
 cout<<"Combinari= "<< p1/(p2*p3);}

```

Observații:

1. Variabilele de memorie **n** și **k** din programul principal, a căror valoare este transmisă ca parametru actual în apelurile de funcție, sunt de același tip ca și parametru formal **n** din antetul funcției operand. În același mod, variabilele de memorie **p1**, **p2** și **p3** din programul principal, a căror valoare este transmisă pentru al doilea parametru actual din apelurile de procedură, sunt de același tip ca și variabila de memorie **p** folosită în subprogram și care reprezintă al doilea parametru formal din antetul procedurii.
2. Activarea subprogramului se face printr-o instrucțiune procedurală. Numele subprogramului apare în două situații: în antetul funcției și în cele trei apeluri ale funcției procedurale (instrucțiuni procedurale).

Programul obținut prin folosirea funcției operand este:

```

#include<iostream.h>
unsigned long fact(int n)
{int i; unsigned long p=1;
 for(i=1;i<=n;i++) p=p*i;
 return p;}
void main()
{int n,k; cout<<"n= "; cin>>n; cout<<"k= "; cin>>k;
 cout<<"Combinari= "<< fact(n)/(fact(k)*fact(n-k)); }

```

Observații:

1. Variabilele de memorie **n** și **k** din programul principal, a căror valoare este transmisă ca parametru actual în apelurile de funcție, sunt de același tip ca și parametru formal **n** din antetul funcției operand. În același mod, variabila de memorie **p** din subprogram, a cărei valoare este returnată prin numele funcției operand, este de același tip ca și rezultatul funcției precizat în antetul funcției.
2. Activarea subprogramului se face în cadrul unei instrucțiuni de atribuire în care numele lui apare ca operand. În cadrul programului, numele subprogramului apare în două situații: în antetul funcției și în cele trei apeluri ale funcției operand (în instrucțiunea de afișare a rezultatului expresiei prin care se calculează valoarea combinărilor).
3. **Comparație.** În varianta funcției procedurale, în funcția rădăcină sunt necesare în plus trei variabile **p1**, **p2** și **p3**. În schimb, în varianta funcției operand, în subprogram se folosește în plus variabila locală **p**.



Comparați, pentru fiecare problemă, cele două variante de implementare a subprogramului. Precizați, pentru fiecare caz, ce tip de subprogram este mai bine să folosiți. Justificați răspunsul.



Recomandări:

1. Se recomandă **folosirea funcțiilor operand** atunci când subprogramul furnizează un singur rezultat.
2. Se recomandă **folosirea funcțiilor procedurale** atunci când subprogramul nu furnizează nici un rezultat sau furnizează mai multe rezultate.

1.1.9. Tablourile de memorie și subprogramele

Dacă parametrii de ieșire sau de intrare-ieșire ai unui subprogram sunt de tip vector, nu trebuie să folosiți transferul prin referință, deoarece identificatorul unui tablou de memorie este o adresă de memorie. Prin urmare, dacă se folosește transferul prin valoare, în stivă se va transfera adresa vectorului, și orice modificare făcută în vector va fi vizibilă și din modulul apelant.

Exemplul 1 – Se citesc de la tastatură elementele de tip întreg a doi vectori, **a** și **b**, care au aceeași dimensiune, **n**. Elementele vectorilor sunt de tip **int**. Se va obține vectorul **c** prin adunarea elementelor celor doi vectori: **a** și **b**. Se sortează crescător vectorul **c** și apoi se afișează. Se folosesc următoarele subprograme: **citeste()** pentru a citi elementele unui vector, **scrie()** pentru a afișa elementele unui vector, **sort()** pentru a sorta elementele unui vector și **aduna()** pentru a aduna elementele a doi vectori.

```
#include <iostream.h>

void citeste(int x[], int n)
{for (int i=0;i<n;i++) {cout<<"x("<<i+1<<"= "; cin>>x[i];}}

void afiseaza(int x[], int n)
{for (int i=0;i<n;i++) cout<<x[i]<<" ";}

void aduna(int x[], int y[], int z[], int n)
{for (int i=0;i<n;i++) z[i]=x[i]+y[i];}

void sort(int x[],int n)
{for (int i=0;i<n-1;i++)
    for (int j=i+1;aux;j<n;j++)
        if (x[i]>x[j]) {aux=x[i]; x[i]=x[j]; x[j]=aux;}}

void main()
{int a[20],b[20],c[20],n;
cout<<"numarul de elemente "; cin>>n;
cout<<"primul vector"<<endl; citeste(a,n);
cout<<"al doilea vector"<<endl; citeste(b,n);
aduna(a,b,c,n); sort(c,n);
cout<<"vectorul rezultat, sortat"<<endl; afiseaza(c,n);}
```

Atenție

Atunci când transmiteți un vector ca parametru nu trebuie să precizați lungimea fizică a vectorului la declararea parametrului. Parantezele pătrate care urmează după numele unui vector informează compilatorul că acest parametru este un vector.

Dacă subprogramul este folosit numai pentru un vector, acesta poate fi declarat ca variabilă globală.

Exemplul 2 – Se citesc de la tastatură cele **n** elemente de tip întreg ale unui vector și o valoare întreagă **x**. Să se afișeze de câte ori apare, în vector, valoarea **x**. Se folosesc următoarele subprograme: **citeste()** pentru a citi elementele vectorului și **numar()** pentru a număra aparițiile valorii **x** în vector.

```
#include <iostream.h>
int a[100],n,x;
void citeste()
{for (int i=0;i<n;i++) {cout<<"a("<<i+1<<"= "; cin>>a[i];}}
```

```

int numar()
{for(int i=0,k=0;i<n;i++) k+=(x==a[i]);
 return k;}

void main()
{cout<<" n="; cin>>n; cout<<" x="; cin>>x; citeste();
 cout<<"Numarul "<<x<<" a fost gasit in vectorul citit de ";
 cout<<numar()<<" ori";}

```

Temă

1. Pentru un vector cu elemente întregi, scrieți câte un subprogram care calculează:
 - a. suma elementelor vectorului;
 - b. suma elementelor pozitive din vector;
 - c. media aritmetică a elementelor vectorului;
 - d. media aritmetică a elementelor pozitive din vector.
2. Scrieți un subprogram care construiește (fără a afișa) vectorul v care conține, în ordine descrescătoare, cele mai mici n numere naturale pare. De exemplu, pentru $n=7$, vectorul v va conține valorile 12, 10, 8, 6, 4, 2, 0. Valoarea lui n ($n < 100$) se transmite ca parametru, vectorul v fiind și el parametru al subprogramului.
(Bacalaureat – Simulare 2004)
3. Funcția **sum** primește prin intermediul parametrului v un vector de numere reale cu 50 de componente și prin intermediul parametrului k un număr natural nenul ($1 < k < 50$). El returnează suma tuturor elementelor v_i ale vectorului, cu proprietatea că $i \leq k$.
 - a. Scrieți definiția completă a subprogramului **sum**.
 - b. Scrieți programul care citește de la tastatură un șir s de 50 de numere reale și apoi două numere naturale n și m ($1 < m < n < 50$) și afișează suma elementelor din șir cu indicii cuprinși între m și n (inclusiv m și n) folosind apeluri ale funcției **sum**.
(Bacalaureat – Sesiunea iunie - iulie 2004)

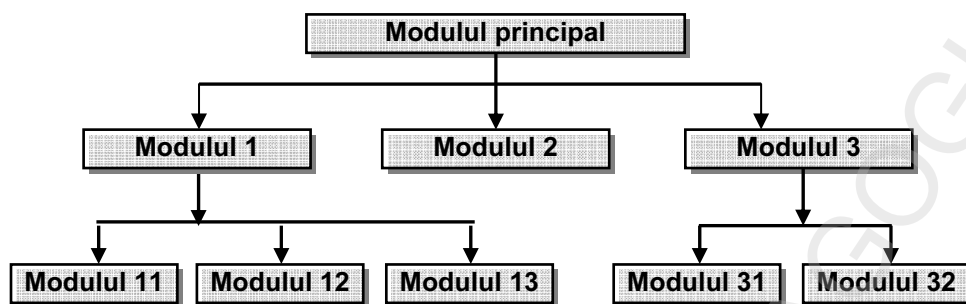
1.1.10. Dezvoltarea programelor

Tehnica **top-down** este o tehnică de proiectare a algoritmilor prin **rafinare iterativă**. Ea constă în descompunerea problemei inițiale în subprobleme, care la rândul lor vor fi supuse aceluiași proces de descompunere, până când se obțin subprobleme cu rezolvare imediată. Folosirea acestei tehnici are următoarele avantaje:

- **Atenția programatorului** se poate concentra la un moment dat numai asupra unei singure subprobleme, și nu asupra întregului ansamblu de subprobleme care formează problema pe care trebuie să o rezolve.
- **Se favorizează lucrul în echipă**, deoarece fiecare subproblemă va putea fi rezolvată de câte un programator, și nu un singur programator trebuie să rezolve întreaga problemă.

Subproblemele în care este descompusă problema folosind tehnica **top-down** se numesc **module**. Fiecare modul poate fi și el descompus în alte module, până se obțin module cu rezolvare imediată.

Rezolvarea unei subprobleme (modul) se face folosind un subprogram, adică fiecare modul va fi implementat cu ajutorul unui subprogram.



În limbajul C++ nu se poate folosi decât **dezvoltarea ascendentă**, adică subprogramele sunt definite unul după altul.

Observație: Pentru a apela, dintr-un subprogram, orice alt subprogram definit în același fișier sursă (în același program), se vor declara subprogramele înaintea funcției rădăcină **main()**, prin **prototip**, după care se vor defini subprogramele, după funcția rădăcină **main()**.

```

#include<iostream.h>
void sb1(); //Prototipul subprogramului sb1()
void sb2(); //Prototipul subprogramului sb2()
void sb3(); //Prototipul subprogramului sb3()
void main() {sb1(); }
void sb1() //Antetul subprogramului sb1()
{cout<<"Subprogramul 1"<<endl;
 sb2();} //Apelul subprogramului sb2()definit ulterior
void sb2() //Antetul subprogramului sb2()
{cout<<"Subprogramul 2"<<endl;
 sb3();} //Apelul subprogramului sb3()definit ulterior
void sb3() //Antetul subprogramului sb3()
{cout<<"Subprogramul 3"<<endl; }
  
```

Recomandare: Pentru claritatea programului, se recomandă declararea tuturor subprogramelor prin prototip, la începutul fișierului sursă, iar definirea lor, ulterior (eventual, după funcția rădăcină **main()**).

Studiu de caz

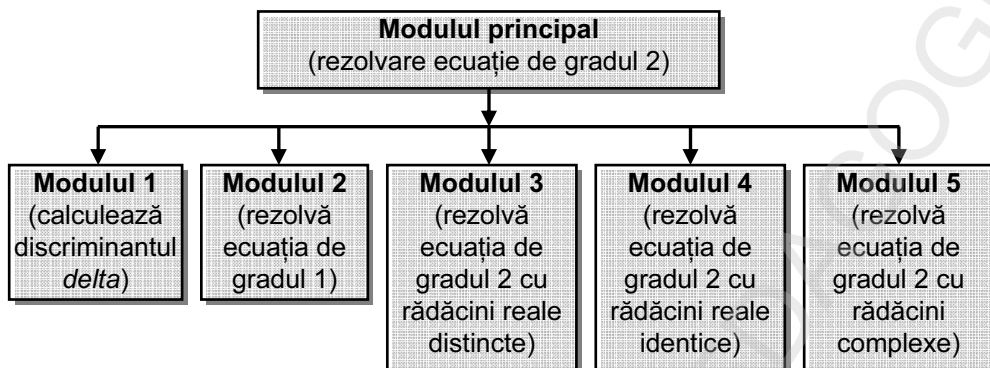
Scop: exemplificarea modului în care poate fi implementat un algoritm cu ajutorul subprogramelor, folosind metoda de proiectare **top-down**.

Enunțul problemei 1: Să se rezolve ecuația de gradul 2: $ax^2+bx+c=0$. Coeficienții ecuației sunt numere întregi.

Folosind tehnica **top-down** problema poate fi împărțită în subprobleme, astfel:

- introducerea coeficienților,
- calcularea discriminantului *delta*,
- rezolvarea ecuației de gradul 1,
- rezolvarea ecuației de gradul 2 cu rădăcini reale distincte,
- rezolvarea ecuației de gradul 2 cu rădăcini reale identice,
- rezolvarea ecuației de gradul 2 cu rădăcini complexe,
- analizarea coeficienților ecuației de gradul 2 și afișarea rezultatelor.

Problema poate fi descompusă în subprobleme care vor fi rezolvate cu ajutorul subprogramelor. **Structura modulelor este următoarea:**



Specificațiile fiecărui modul (subprogram) sunt:

1. Modulul 1 (delta):

- **Date de intrare:** a, b, c – coeficienții ecuației de gradul 2;
- **Date de ieșire:** discriminantul d ;
- **Funcția** modulului: calcularea discriminantului $delta$.

2. Modulul 2 (ec1):

- **Date de intrare:** b, c – coeficienții b și c ai ecuației de gradul 2;
- **Date de ieșire:** nici una;
- **Funcția** modulului: rezolvarea ecuației de gradul 1 și afișarea rezultatului (soluția x sau un mesaj).

3. Modulul 3 (ec2_2r):

- **Date de intrare:** a, b, c – coeficienții ecuației de gradul 2;
- **Date de ieșire:** x_1, x_2 – cele două soluții reale distincte ale ecuației;
- **Funcția** modulului: rezolvarea ecuației de gradul 2 cu soluții reale distincte.

4. Modulul 4 (ec2_1r):

- **Date de intrare:** a, b – coeficienții ecuației de gradul 2;
- **Date de ieșire:** x – cele două soluții reale identice ale ecuației;
- **Funcția** modulului: rezolvarea ecuației de gradul 2 cu soluții reale identice.

5. Modulul 5 (ec2_2c):

- **Date de intrare:** a, b, c – coeficienții ecuației de gradul 2;
- **Date de ieșire:** u, v – partea reală și partea imaginară ale celor două soluții complexe ale ecuației;
- **Funcția** modulului: rezolvarea ecuației de gradul 2 cu soluții complexe.

Pentru rezolvarea subproblemei modulelor 1 și 4 se vor folosi subprograme de tip **funcție operand** ($delta$ și $ec2_1r$), deoarece aceste module furnizează modulului principal un singur rezultat de tip întreg, respectiv real, iar pentru rezolvarea subproblemelor modulelor 2, 3, și 5 se vor folosi subprograme de tip **funcție procedurală**, deoarece ele furnizează modulului principal mai multe rezultate ($ec2_2r$ și $ec2_2c$) sau nici un rezultat ($ec1$). În programul principal (modulul principal) se vor executa următoarele acțiuni:

- Se introduc de la tastatură coeficienții ecuației: a, b, c .
- Se analizează coeficientul a al ecuației.
- Dacă ecuația de gradul 2 a degenerat într-o ecuație de gradul 1, se rezolvă ecuația de gradul 1.

- Dacă ecuația de gradul 2 nu a degenerat într-o ecuație de gradul 1, se analizează discriminantul *delta* – calculat prin funcția `delta()` – și, în funcție de valoarea lui, se apelează subprogramele prin care se rezolvă fiecare caz.

Programul obținut este:

```
#include<iostream.h>
#include<math.h>
void ec1(int,int);
int delta(int,int,int);
void ec2_2r(int ,int ,int ,float &, float &);
float ec2_1r(int ,int );
void ec2_2c(int ,int ,int ,float &, float &);
void main()
{int a,b,c; float x1,x2;
 cout<<"a= "; cin>>a; cout<<"b= "; cin>>b; cout<<"c= "; cin>>c;
 if (a==0) ec1(b,c);
 else
  if (delta(a,b,c)>0)
   {ec2_2r(a,b,c,x1,x2);
    cout<<"Ecuatia are doua radacini reale diferite "<<endl;
    cout<<"x1= "<<x1<<endl; cout<<"x2= "<<x2<<endl;}
  else
   if (delta(a,b,c)==0)
    {cout<<"Ecuatia are doua solutii reale identice "<<endl;
     cout<<"x1=x2= "<<ec2_1r(a,b);}
   else
    {ec2_2c(a,b,c,x1,x2);
     cout<<"Ecuatia are solutii complexe "<<endl;
     cout<<"x1= "<<x1<<" +i*"<<x2<<endl;
     cout<<"x2= "<<x1<<" -i*"<<x2<<endl;}}
void ec1(int b,int c)
{float x;
 if (b==0)
  if (c==0) cout<<"Ecuatia are o infinitate de solutii"<<endl;
  else cout<<"Ecuatia nu are solutii"<<endl;
 else
  {x=-(float)c/b;
   cout<<"Ecuatia a degenerat in ecuatie de gradul I "<<endl;
   cout<<"cu radacina x= "<<x;}}
int delta(int a,int b,int c)
{int d=pow(b,2)-4*a*c;
 return d;}
void ec2_2r(int a,int b,int c,float &x1, float &x2)
{x1=(-b+sqrt(delta(a,b,c)))/(2*a);
 x2=(-b-sqrt(delta(a,b,c)))/(2*a);}
float ec2_1r(int a,int b)
{float x=(float)(-b)/(2*a);
 return x;}
```

```
void ec2_2c(int a,int b,int c,float &u, float &v)
{u=(float) (-b/(2*a)); v=sqrt(abs(delta(a,b,c)))/(2*a);}
```

Temă



Refaceți programul astfel:

- a. Pentru a transmite datele între module, nu folosiți parametrii de comunicație, ci variabilele globale;
- b. Nu folosiți prototipurile funcțiilor (**Atenție!** Deoarece funcția **delta()** este activată în subprogramele **ec2_2r()** și **ec2_2c()** trebuie să o definiți înaintea lor.)

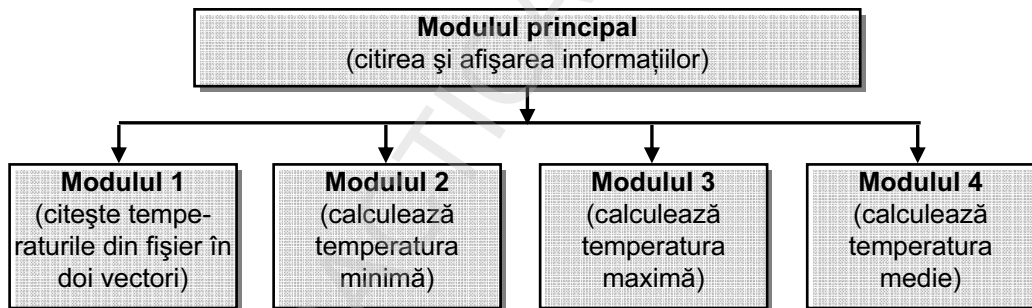
Enunțul problemei 2: În fișierul *meteo.txt* sunt înregistrate, pe mai multe rânduri, perechi de numere întregi separate prin spațiu, care reprezintă temperaturile zilnice – minimă și maximă – din România, într-o perioadă de timp. Să se afișeze temperatura minimă, temperatura maximă și temperatura medie din România din acea perioadă.

Pentru memorarea temperaturilor se folosesc doi vectori: **a** – pentru temperaturile minime și **b** – pentru temperaturile maxime.

Folosind tehnica **top-down** problema poate fi împărțită în subprobleme, astfel:

- crearea celor doi vectori prin citirea datelor din fișier,
- calcularea temperaturii minime,
- calcularea temperaturii maxime,
- calcularea temperaturii medii.

Problema poate fi descompusă în subprobleme, care vor fi rezolvate cu ajutorul subprogramelor. **Structura modulelor este următoarea:**



Specificațiile fiecărui modul (subprogram) sunt:

1. Modulul 1 (citește):

- **Date de intrare:** niciuna;
- **Date de ieșire:** **a** și **b** – vectorii în care se memorează temperaturile minime, respectiv maxime – și **n** – numărul de zile ale perioadei analizate (lungimea logică a vectorilor);
- **Funcția** modului: crearea celor doi vectori prin citirea temperaturilor din fișier.

2. Modulul 2 (min):

- **Date de intrare:** **a** – vectorul în care se memorează temperaturile minime – și **n** – numărul de zile ale perioadei analizate (lungimea logică a vectorului);
- **Data de ieșire:** temperatura minimă din perioada analizată;
- **Funcția** modului: calcularea temperaturii minime din perioada analizată.

3. Modulul 3 (max):

- **Date de intrare:** **b** – vectorul în care se memorează temperaturile maxime – și **n** – numărul de zile ale perioadei analizate (lungimea logică a vectorului);

- **Data de ieșire:** temperatura maximă din perioada analizată;
- **Funcția** modului: calcularea temperaturii maxime din perioada analizată.

4. Modulul 4 (**media**):

5. **Date de intrare:** a și b – vectorii în care se memorează temperaturile minime, respectiv maxime – și n – numărul de zile ale perioadei analizate (lungimea logică a vectorilor);
 - **Data de ieșire:** temperatura medie din perioada analizată;
 - **Funcția** modului: calcularea temperaturii medii din perioada analizată.

Pentru rezolvarea subproblemei modulelor 2, 3 și 4, se vor folosi subprograme de tip **funcție operand** (*min*, *max* și *media*), deoarece aceste module furnizează modulului principal un singur rezultat de tip întreg (*min*, *max*) și de tip real (*media*), iar pentru rezolvarea subproblemei modulului 1 se va folosi un subprogram de tip **funcție procedurală** (*citește*), deoarece el furnizează modulului principal mai multe rezultate (vectorul cu temperaturile minime, respectiv maxime, și lungimea logică a vectorului). În programul principal (modulul principal) se vor executa următoarele acțiuni:

- Se creează cei doi vectori, prin citirea datelor din fișier, apelându-se subprogramul `citește()`.
- Se afișează temperatura minimă apelându-se subprogramul `min()`.
- Se afișează temperatura maximă apelându-se subprogramul `max()`.
- Se afișează temperatura medie apelându-se subprogramul `media()`.

Programul obținut este:

```
#include <fstream.h>

void citește(int &, int a[], int b[]);
int min(int n, int b[]);
int max(int n, int b[]);
float media(int n, int a[], int b[]);

void main()
{int a[50], b[50], n;
  citește(n, a, b);
  for (int i=0; i<n; i++) cout<<a[i]<<" "<<b[i]<<endl;
  cout<<"Temperatura minima= "<<min(n, a)<<endl;
  cout<<"Temperatura maxima= "<<max(n, b)<<endl;
  cout<<"Temperatura medie= "<<media(n, a, b)<<endl; }

void citește(int &n, int a[], int b[])
{fstream f("meteo.txt", ios::in);
  int i=0;
  while (f>>a[i]>>b[i]) i++;
  n=i; f.close(); }

int min(int n, int a[])
{int m=a[0];
  for (int i=1; i<n; i++)
    if (a[i]<m) m=a[i];
  return m; }

int max(int n, int b[])
{int m=b[0];
```

```
for (int i=1;i<n;i++) if (b[i]>m) m=b[i];
return m;}

float media(int n, int a[], int b[])
{int s=0;
for (int i=0;i<n;i++) s=s+a[i]+b[i];
return (float)s/(2*n);}
```

Temă



1. Refaceți programul, astfel:

- a. Nu folosiți prototipurile funcțiilor.
- b. Pentru a transmite datele între module, nu folosiți parametri de comunicație, ci variabilele globale.

2. Calculați suma:

$$s=1-x+x^2/2!-x^3/3!+x^4/4!-x^5/5!+...+(-1)^{2n+1}x^n/(2n+1)!$$

unde x și n sunt două numere naturale care se citesc de la tastatură.

Descompuneți problema în subprobleme. Rezolvați fiecare subproblemă cu ajutorul unui subprogram.



1.1.11. Subprogramele de sistem

Compilerul C++ vă pune la dispoziție o bibliotecă cu sute de funcții, pe care le puteți folosi pentru a rezolva o anumită sarcină. Pentru a putea folosi o funcție de sistem, trebuie să știți să interpretați prototipul funcției pe care vi-l pune la dispoziție autodocumentarea (**help**-ul) limbajului de programare.

Exemplul 1

Funcția **sqrt()** din fișierul antet **math.h** furnizează, prin numele ei, radical de ordinul 2 din parametrul x . Ea are prototipul:

```
double sqrt(double x);
```

Interpretați prototipul funcției, astfel:

- Rezultatul funcției este de tip real – **double**.
- Funcția are un singur parametru de tip real – **double**.

Exemplul 2

Funcția **poly()** din fișierul antet **math.h** furnizează, prin numele ei, valoarea unui polinom de gradul *degree*, cu coeficienții *coefs*, pentru x precizat. Ea are prototipul:

```
double poly(double x, int degree, double coefs[]);
```

Interpretați prototipul funcției, astfel:

- Rezultatul funcției este de tip real – **double**.
- Funcția are trei parametri: unul de tip real, pentru x – **double**, unul de tip întreg, pentru gradul polinomului – **int**, și unul de tip vector de numere reale, pentru coeficienții polinomului – **double []**.

Atenție

Conflictele de nume dintre o funcție de sistem și o funcție utilizator se rezolvă astfel: dacă ați creat o funcție utilizator, căreia i-ați atribuit același nume cu al unei funcții de sistem, compilerul va alege funcția utilizator.

Funcții de sistem utile

Funcții matematice – fișier antet **math.h**

Funcția	Tip rezultat	Tip parametri	Furnizează	Exemplu
abs(x)	int	int	modulul lui x	abs(-12) → 12
fabs(x)	double	double		fabs(-1.2) → 1.2
floor(x)	double	double	cel mai apropiat întreg mai mic sau egal cu x	floor(11.5) → 11
floorl(x)	long double	long double		floor(-2.7) → -3
ceil(x)	double	double	cel mai apropiat întreg mai mare sau egal cu x	ceil(11.5) → 12
ceil(x)	long double	long double		ceil(-2.7) → -2
sin(x)	double	double	sinus de x	sin(0.5) → 0.479426
cos(x)	double	double	cosinus de x	cos(0.5) → 0.877583
tan(x)	double	double	tangentă de x	tan(0.5) → 0.546302
sqrt(x)	double	double	radical de ordinul 2 din x	sqrt(9) → 3
pow(x,y)	double	double	x la puterea y	pow(2,4) → 16
pow10(x)	double	int	10 la puterea x	pow10(2) → 100

Funcții utile pentru operațiile de citire și scriere – fișierul antet **conio.h**

- clrscr()** Șterge informațiile de pe ecran. Această funcție se scrie la începutul programului, ca pe ecran să fie afișate numai rezultatele obținute în urma executării programului.
- getch()** Așteaptă introducerea unui caracter de la tastatură, pe care nu-l va afișa pe ecran. Rezultatul furnizat este de tip **int** și este codul caracterului introdus.

Funcții pentru generarea numerelor aleatoare – fișier antet **stdlib.h**

- randomize()** Inițializează generatorul de numere aleatoare cu un număr aleator.
- rand()** Furnizează un rezultat de tip **int** care este un număr aleator cuprins între 0 și 32767.

Exemplu – Se simulează aruncarea unui zar. Pentru a genera un număr cu valoarea cuprinsă între 1 și 6, se generează aleator un număr, cu funcția **rand()**, și se calculează restul împărțirii acestui număr la 6 (care poate lua o valoare de la 0 la 5), la care se adaugă o unitate.

```
#include<iostream.h>
#include <stdlib.h>
void main()
{randomize(); //Se inițializează generatorul de numere aleatoare.
  cout<<rand()%6+1<<" ";}
//Se generează aleator un număr cu valoarea cuprinsă între 1 și 6.
```



1. Precizați tipul funcției **rand()**. Dați un exemplu de apel pentru această funcție.
2. Precizați tipul funcției **fabs()**. Dați un exemplu de apel pentru această funcție. Precizați tipul parametrului.

Evaluare

Răspundeți:

1. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int a,b,c;
void sb(int &b){cout<<a<<" "<<b<<" "<<c;}
void main()
{cout<<a<<" "<<b<<" "<<c; a=10; b=20; c=30; sb(a);
  cout<<a<<" "<<b<<" "<<c;}
```

2. Ce se va afișa în urma execuției următoarei secvențe de program?

```
float x;
void sb(int x) {x=1;}
void main() {x=2.5; sb(x); cout<<x;}
```

3. Ce se va afișa în urma execuției următoarei secvențe de program?

```
float x;
void sb1(int &x) {cout<<x; x=1; cout<<x;}
void main() {x=2.5; sb(x); cout<<x;}
```

4. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int x;
void sb1(int x) {x=10; cout<<x; }
void sb2(int x) {x=20; cout<<x; }
void main() {x=30; sb1(x); cout<<x; sb2(x); cout<<x;}
```

5. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int a,b;
int sb(int a)
{int i=0; while (a!=0) {a/=10; i++;}
cout<<b; return i;}
void main() {a=12345; b=sb(a); cout<<b;}
```

6. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int x;
void sb1(int x) {x=20; cout<<x; }
void sb2(int &x) {x=30; cout<<x; }
void main() {x=50; cout<<x; sb1(x); cout<<x; sb2(x); cout<<x;}
```

7. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int x;
void sb1(int); void sb2(int &);
void sb1(int x) {x=20; sb2(x);}
void sb2(int &x) {x=30; sb1(x);}
void main() {x=50; cout<<x; sb1(x); cout<<x; sb2(x); cout<<x;}
```

8. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int a,b;
void sb1(int x, int y) {x=x-y; y=y-x;}
void sb2(int x, int y) {a=sb1(x,y); b=sb1(y,x);}
void main() {a=1; b=2; sb1(a,b); cout<<a<<b<<; sb1(b,a); cout<<a<<b;}
```

9. Ce se va afișa în urma execuției următoarei secvențe de program?

```
int sb(int a)
{int i=0; while (a!=0) {int i=10; a/=2; i++;}
return i;}
void main() {cout<<sb(23);}
```

Adevărat sau Fals:

1. Antetul de funcție `int f(int x; int y);` este corect.
2. Antetul de funcție `f(int x, int y)` nu este corect.

3. Antetul de funcție `void f(int x[25], int &n)` este corect.
4. La apelul unei funcții, parametri actuali sunt înlocuiți cu parametri formali.
5. Următorul program produce eroare la compilare:

```
#include <iostream.h>
#include <math.h>
int d(int a, int b){return sqrt(a*a+b*b);}
void main() {cout<<d(5,4);}
```

6. Următorul program nu produce eroare la compilare:

```
#include <iostream.h>
#include <math.h>
float mg(int x, int y, float & mg)
{mg=sqrt(x*y); return sqrt(x*y);}
void main() {float x; cout<<x<<" "<<mg(3,4,x); }
```

7. Următorul program produce eroare la compilare:

```
#include <iostream.h>
int f(int x, int y)
{for (int i=x;i<y;i++) if(i>x+1 && i<y-1) return i;
return x+y; }
void main() {cout<<f(1,5); }
```

8. Următorul program produce eroare la compilare:

```
#include <iostream.h>
int f(int x=20, int y=10) {return x/y;}
void main() {cout<<f(15,5)<<" "<<f(15)<<" "<<f(); }
```

Alegeți:

1. Care dintre următoarele anteturi de subprogram este un antet corect pentru o funcție reală cu parametru întreg?
 - a. `double f(long x);`
 - b. `double f(int x)`
 - c. `long double f(int x)`
 - d. `float f(int x);`
2. Care dintre următoarele anteturi de subprogram sunt corecte?
 - a. `int f(int x);`
 - b. `f(int x, int y, char z)`
 - c. `int f(int x,y)`
 - d. `float f(int x; int y)`
3. Funcția `m()` are doi parametri reali și furnizează cea mai mică valoare dintre cei doi parametri. Care dintre următoarele instrucțiuni afișează cea mai mică valoare dintre `a`, `b` și `c`?
 - a. `cout<<m(a,b,c);`
 - b. `cout<<m(m(a,b),m(b,c));`
 - c. `cout<<m(a,m(b,c));`
 - d. `cout<<m(m(a,b),c);`
4. Se consideră că următoarea funcție furnizează cel mai mare divizor comun a două numere transmise ca parametru.

```
int cmmdc(int a, int b)
{while (...) if (a>b) a-=b; else b-=a;
return a;}
```

Ce condiție trebuie scrisă în instrucțiunea `while`?

- a. `a>b`
 - b. `a!=b`
 - c. `a==b`
 - d. `a<b`
5. Se consideră că următoarea funcție testează dacă numărul transmis ca parametru este un număr prim.


```
int prim(int n)
{int i=3;
  if (n%2) while (...) if (n%i==0) return 0; else i+=2;
  else return 0;
return 1;}
```

Ce condiție trebuie scrisă în instrucțiunea **while**?

- a. $i < \sqrt{n}$ b. $i < n/2$ c. $i \leq \sqrt{n}$ d. $i \leq n/2$

6. Pentru fiecare antet al subprogramului **sb** din coloana A, există în coloana B valorile care vor fi afișate pe ecran. Alegeți atribuirile corecte:

```
int a,b;
void sb(...) {int a; a=m; n+=a; m=n;}
void main() {a=10; b=20; sb(a,b); cout<<a<<" "<<b<<" ";
```

A	B
A1. void sb(int m, int n)	B1. 30, 30
A2. void sb(int &m, int n)	B2. 10, 30
A3. void sb(int m, int &n)	B3. 10, 20
A4. void sb(int &m, int &n)	B4. 30, 20
a. A1 – B4; A2 – B1; A3 – B3; A4 – B2;	
b. A1 – B3; A2 – B1; A3 – B4; A4 – B2;	
c. A1 – B4; A2 – B1; A3 – B2; A4 – B3;	
d. A1 – B1; A2 – B4; A3 – B2; A4 – B3;	

7. Se consideră următorul program:

```
int a,b;
void sb(int &a, int b) {a++; b--; a++;}
void main()
{a=5; b=10; sb(a,b); cout<<a<<" "<<b<<" ";
```

Ce valori se vor afișa pe ecran?

- a. 5, 10, 10, 20 b. 5, 9, 10, 19 c. 7, 10, 12, 20 d. 7, 9, 12, 19

8. Care dintre următoarele subprograme adună două valori întregi transmise ca parametri?

- a. `int s(int x, int y)`
`{int z=x; z+=y; return z;}`
b. `void s(int x, int y, int &z)`
`{z=x+y;}`
c. `int s(int &x, int &y)`
`{s=y; s=s+x; }`
d. `void s(int x, int y)`
`{int z=y+x; return z;}`

9. Care dintre următoarele subprograme realizează interschimbarea valorilor a două numere întregi transmise ca parametri?

- a. `void s(int x, int y)`
`{int z=y; y=x; x=y;}`
b. `void s(int x, int y)`
`{ x=y+x; x=y-x; y=y-x;}`
c. `void s(int &x, int &y)`
`{x=x-y; y=x+y; x=y-x;}`
d. `void s(int &x, int &y)`
`{ x=y+x; x=y-x; y=y-x;}`

10. Subprogramul **z1** are un parametru întreg și returnează o valoare reală. Care este antetul corect al subprogramului **z1** ?

- a. `int z1(float n)` b. `float z1(int n)`
c. `void z1(float n)` d. `void z1(float &n; float r)`

(Bacalaureat – Sesiunea august 2004)

11. Subprogramul **intersch** realizează interschimbarea valorilor a două variabile întregi transmise prin intermediul parametrilor formali **x** și **y**. Antetul subprogramului este:
- a. `int intersch(int &x, &y)` b. `void intersch(int x, int y)`
c. `void intersch(int &x, int &y)` d. `int intersch(int x)`
(Bacalaureat – Simulare 2003)
12. Care dintre următoarele subprograme returnează cea mai mică valoare dintre două numere întregi transmise ca parametri?
- a. `int m(int &x, int &y)` b. `void m(int &x, int &y)`
 `{m=x; if(y<x) m=y;` `{if(y<x) m=y;`
 `return m;}` `else m=x;}`
c. `int m(int x, int y)` d. `int m(int x, int y)`
 `{if(y<x) x=y;` `{if(y<x) return y;`
 `return x;}` `else return x;}`
13. Subprogramul **cifre** calculează numărul **i** de cifre ale unui număr natural **n** transmis ca parametru și construiește vectorul **v** format din cifrele lui **n**. Care este antetul corect al unui astfel de subprogram?
- a. `void cifre(long n, vector v, int &i)`
b. `void cifre(long n, int i, vector v)`
c. `void cifre(long n; vector v; int &i)`
d. `void cifre(long &n; vector v; int i)`
(Bacalaureat – Sesiunea iunie - iulie 2003)
14. Dacă **a** este o variabilă globală și la începutul subprogramului **sub** este definită o variabilă locală **a**, atunci în instrucțiunea **a=a+1** din subprogramul **sub**, **a** se referă la:
- a. variabila globală **a** b. nu se poate defini **a** ca variabilă globală și variabilă locală
c. variabila locală **a** d. unul la variabila locală, iar altul la variabila globală
(Bacalaureat – Sesiunea iunie - iulie 2004)
15. Se presupune că este definită o funcție **min** care primește două valori reale prin intermediul a doi parametri și returnează cea mai mică dintre cele două valori. Stabiliți care dintre următoarele expresii este egală cu cea mai mare dintre valorile reale **a** și **b**.
- a. $\min(a,b) - a - b$ b. $a - \min(a,b) + b - \min(a,b)$
c. $a + b - \min(a,b)$ d. $\min(a,b)$
(Bacalaureat – Simulare 2003)
16. Este definită o funcție **min** care primește două valori reale prin intermediul a doi parametri și returnează cea mai mică dintre cele două valori. Stabiliți care dintre următoarele expresii **nu** este egală cu cea mai mică dintre valorile reale **a**, **b** și **c**.
- a. $a + b + c - \min(a,b) - \min(a,c)$ b. $\min(\min(a,b), \min(a,c))$
c. $\min(\min(a,b), c)$ d. $\min(a, \min(b,c))$
(Bacalaureat – Sesiune specială 2003)
17. Este definită funcția **max** care primește două valori întregi prin intermediul parametrilor formali **a** și **b** (primul fiind **a** și al doilea **b**) și returnează cea mai mare cifră din șirul infinit al zecimalelor raportului **a/b**. Astfel, **max(5,12)=6** deoarece $5/12=0.41666\dots$, iar **max(1,8)=5** deoarece $1/8=0.125000\dots$. Stabiliți care dintre următoarele expresii este adevărată dacă și numai dacă **m** este divizor al lui **n**.
- a. $\max(n,m) \leq 0$ b. $\max(m,n) == 0$ c. $\max(m,n) == 0 > 0$ d. $\max(n,m) != 0$
(Bacalaureat – Sesiune iunie - iulie 2003)

18. Se știe că există o variabilă globală **v** de tip vector și că vectorul **v** este format numai din 0 și 1. Știind că subprogramul alăturat returnează numărul de componente nenule aflate pe primele **n** poziții ale vectorului **v**, stabiliți ce este incorect în definiția acestuia.

- a. returnează o valoare nedeterminată
- b. nu este corect antetul funcției
- c. totul este corect
- d. este incorectă instrucțiunea **for**

```
int f(int a)
{int k=0;
 for (int i=0;i<n;i++)
  k+=v[i];}
```

(Bacalaureat – Sesiune iunie - iulie 2003)

19. În subprogramul alăturat, **i** este:

- a. parametru de intrare
- b. variabilă globală
- c. parametru de ieșire
- d. variabilă locală

```
void fa(unsigned int x)
{unsigned int i;
 for (i=1;i<=n;i++)
  cout<<i<<" ";}
```

(Bacalaureat – Simulare 2004)

20. Subprogramul **scif** returnează suma cifrelor unui număr natural transmis ca parametru. Stabiliți valoarea expresiei **scif(scif(518) + scif(518))**.

- a. 14
- b. 10
- c. 28
- d. 1

(Bacalaureat – Simulare 2005)

21. Subprogramul alăturat **z1** are un singur parametru real **a**. Știind că secvența **float x=4.25;**
cout<<x<<' '<<z1(r);, afișează 4.25 2, stabiliți care este antetul corect al subprogramului **z1**?

```
...../*antet*/
{a=a*10;
 return (int)a%10;}
```

- a. **void z1(float &a)**
- b. **void z1(float a)**
- c. **int z1(float &a)**
- d. **int z1(float a)**

(Bacalaureat – Sesiunea iunie - iulie 2004)

22. Pentru valori strict pozitive ale parametrului **a**, funcția **f** definită alăturat returnează valoarea 1 dacă și numai dacă valoarea lui **a** este un număr natural care:

- a. are ultima cifră mai mică sau egală cu 5
- b. are cel puțin o cifră mai mică sau egală cu 5
- c. are prima cifră mai mică sau egală cu 5
- d. are cel mult o cifră mai mică sau egală cu 5

```
int f(int a)
{while (a%10>5) a/=10;
 return a>0;}
```

(Bacalaureat – Sesiunea specială 2004)

23. Pentru subprogramul următor, variabila întregă **x** și variabila reală **r=4.25**, stabiliți câte dintre secvențele alăturate afișează valoarea 4.

int z1(float &a)	x=z1(r); cout<<x+z1(r);
{if (a<0) a=-a;	x=z1(r); cout<<z1(r)+x;
a*=10;	cout<<z1(r)+z1(r);
return (int)a%10;	x=z1(r); cout<<x+x;

- a. 4
- b. 3
- c. 2
- d. 1

(Bacalaureat – Sesiunea iunie–iulie 2004)

24. Pentru variabilele **n**, **i** și **j** întregi, secvența alăturată afișează cel mai mare și cel mai mic divizor propriu al numărului natural neprim citit (1 și **n** sunt divizori improprii). Care este antetul corect pentru subprogramul **divi**?

```
cin>>n;
divi(n,i,j);
cout<<i<<j;}
```

- a. **int divi(int &n, int i, int j)**
- b. **int divi(int n, int &i, int j)**
- c. **void divi(int n, int i, int j)**
- d. **void divi(int n, int &i, int &j)**

(Bacalaureat – Sesiunea iunie–iulie 2004)

25. Este definită o funcție **smax** care primește două valori întregi prin intermediul a doi parametri și returnează suma tuturor cifrelor celor două numere. De exemplu, **smax(73, 608)** returnează 24. Stabiliți în ce mod se poate apela funcția **smax** pentru a determina suma cifrelor unui număr întreg **n**.

- a. **smax(n, n)** b. **smax(n, 0)** c. **smax(n, 1)** d. nu se poate utiliza
(Bacalaureat – Sesiune specială 2003)

Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra în **echipă**. Fiecare miniproiect va conține:

- descompunerea problemei în subprobleme și rezolvarea fiecărei subprobleme cu ajutorul unui subprogram;**
 - specificările fiecărui modul (subprogram);
 - explicarea noțiunilor teoretice folosite pentru realizarea subprogramelor;
 - alte trei exemple de probleme în care se vor folosi subprograme create pentru rezolvarea problemei inițiale.
- Se citesc de la tastatură **n** numere naturale. Să se afișeze numerele care au cea mai mare sumă a divizorilor proprii.
 - Se citește de la tastatură un număr natural **n**. Să se afișeze **numerele rotunde** mai mici decât **n**. Un număr rotund este un număr care, reprezentat în binar, are același număr de cifre de 0 și de 1.
 - Se citesc de la tastatură două numere naturale **n** și **m**, care reprezintă numărul de elemente a doi vectori cu numere întregi, **a** și **b**. Elementele celor doi vectori se citesc de la tastatură. Afișați câte dintre componentele vectorului **a** sunt strict mai mici decât componentele vectorului **b**.
 - Se citesc de la tastatură două mulțimi de numere întregi. Afișați intersecția, reuniunea și diferența dintre cele două mulțimi.
 - Se citesc de la tastatură **n** numere naturale. Să se verifice dacă numărul format cu prima cifră a fiecărui număr este palindrom.
 - Se citesc de la tastatură: un număr **q** ($2 \leq q \leq 9$), care reprezintă o bază de numerație, un număr **c**, care reprezintă o cifră din această bază de numerație, și **n** numere naturale. Să se afișeze numerele care au cele mai multe cifre **c**, atunci când sunt reprezentate în baza **q**.
 - Se citește de la tastatură o mulțime de **n** numere naturale. Să se elimine din această mulțime numerele palindrom care au cel mai mare număr de cifre.
 - Într-o fabrică lucrează **n** muncitori. În funcție de calificare, fiecărui muncitor *i* se atribuie o categorie salarială. Există 5 categorii salariale: **p1**, **p2**, **p3**, **p4** și **p5** (valori reale cuprinse între 1 și 3 – de exemplu, 1; 1,5; 2; 2,5 și 3) și un salariu tarifar de bază, pe oră. Într-un vector se memorează orele lucrate de muncitori. Într-un alt vector, se memorează categoria salarială a fiecărui muncitor. Salariul brut al unui muncitor este dat de produsul dintre numărul de ore lucrate, salariul tarifar pe oră și categoria salarială. Salariul net se obține prin deducerea impozitului de 16% din salariul brut. Numărul **n**, valorile din cei doi vectori, valorile pentru categoriile salariale **p1**, **p2**, **p3**, **p4** și **p5** și salariul tarifar orar – se citesc de la tastatură. Afișați salariul brut și salariul net ale fiecărui muncitor și totalul salariilor brute, pentru toți cei **n** muncitori.
 - Într-un fișier text sunt scrise pe un rând, separate prin spațiu, **n** numere reale care reprezintă înălțimea unor elevi. Fiecare elev se identifică prin numărul de ordine, la citirea înălțimii din fișier. Să se afișeze ordinea în care trebuie aranjați crescător, după înălțime, cei **n** elevi, elevul care este cel mai înalt și elevul care este cel mai scund.

1.2. Recursivitatea

1.2.1. Definiția procesului recursiv

Un proces poate fi descris printr-un subprogram.

Procesul recursiv este procesul care, în timpul execuției, generează apariția unor procese similare lui, aflate în legătură directă cu procesul care le generează.

Exemplu de proces recursiv: Sunteți într-o localitate necunoscută și căutați clădirea muzeului. Întrebați prima persoană întâlnită. Traseul fiind complicat, vă explică în ce fel să ajungeți la strada următoare de pe traseul către muzeu. Aici veți întreba următoarea persoană cum să ajungeți la muzeu și ea vă va îndruma la strada următoare de pe traseu ș.a.m.d. Fiecare persoană pe care o întrebați vă furnizează o soluție care vă apropie de muzeu, astfel încât ultima persoană chestionată vă va arăta clar unde este muzeul. Această identificare a traseului este un proces recursiv. Fiecare chestionare a unei persoane generează același tip de proces: deplasare pe o porțiune a traseului și o nouă chestionare.

Se consideră că o noțiune este definită recursiv dacă, în cadrul definiției, apare însăși noțiunea care se definește. În exemplul de mai sus, traseul este definit recursiv, printr-un nou traseu către muzeu (traseu care pornește din locația în care vă găsiți la un moment dat).

Considerăm fraza următoare: „*Sportul înseamnă să faceți mișcare, iar mișcarea înseamnă să faceți sport.*”. Chiar dacă noțiunile *sport* și *mișcare* se autodefinesc printr-un proces recursiv, fraza nu furnizează nici un fel de informație – ori considerăm că „*sportul înseamnă să faceți sport*”, ori „*mișcarea înseamnă să faceți mișcare*”.

Procesele recursive pot fi:

- **Finite.** Încheierea execuției procesului se face după un număr determinat de operații executabile (în cazul modelării lor cu ajutorul subprogramelor, după un număr determinat de instrucțiuni executabile). În exemplul anterior, veți ajunge la muzeu după ce ați chestionat un număr determinat de persoane, deci procesul este finit.
- **Infinite.** Sunt opuse proceselor finite. În exemplul anterior, definiția *sportului* este un proces infinit, deoarece ea nu se oprește după un număr determinat de definiții.

Procesele descrise prin subprograme sunt procese finite. De aceea, trebuie definită **condiția de terminare**. Prin această condiție, se descrie modul prin care un proces finit nu devine un proces infinit.

De exemplu, dacă vreți să calculați modulul unui număr, puteți să generați un proces recursiv, astfel: dacă numărul este pozitiv, modulul este egal cu numărul; altfel, este egal cu modulul din valoarea numărului înmulțit cu -1 (pentru ca valoarea să devină pozitivă). Astfel, **mod(5)=5**, iar **mod(-5)=mod(-(-5))=mod(5)=5**. **Condiția de terminare** a procesului recursiv este ca variabila *x* să fie pozitivă. Funcția matematică poate fi implementată cu ajutorul următorului program:

$$\text{mod}(x) = \begin{cases} x & \text{pentru } x > 0 \\ \text{mod}(-x) & \text{pentru } x \leq 0 \end{cases}$$

```
#include <iostream.h>
int modul(int x)
{if(x>=0) return x; else return modul(-x);}
void main()
{int x; cout<<"x= "; cin>>x; cout<<modul(x);}
```

Puteți să generați un proces recursiv pentru calculul valorii maxime dintre două numere a și b , astfel: dacă $a > b$, maximul este valoarea primului număr, adică a ; altfel, este egal cu maximul dintre b și a . Astfel, $\max(5,2)=5$, iar $\max(2,5)=\max(5,2)=5$. **Condiția de terminare** a procesului recursiv este ca variabila a să fie mai mare decât variabila b . Funcția matematică poate fi implementată cu ajutorul următorului program:

$$\max(a,b) = \begin{cases} a & \text{pentru } a > b \\ \max(b,a) & \text{pentru } a \leq b \end{cases}$$

```
#include <iostream.h>
int max(int a,int b)
{if (a>b) return a; else return max(b,a);}
void main()
{int a,b; cout<<"a= "; cin>>a; cout<<"b= "; cin>>b;
cout<<max(a,b);}
```

Temă



Scrieți un program în care se citesc două numere de la tastatură, a și b , și se afișează valoarea minimă dintre cele două numere. Valoarea minimă se va calcula cu ajutorul unei funcții în care formula de calcul este definită printr-un proces recursiv.

Studiu de caz

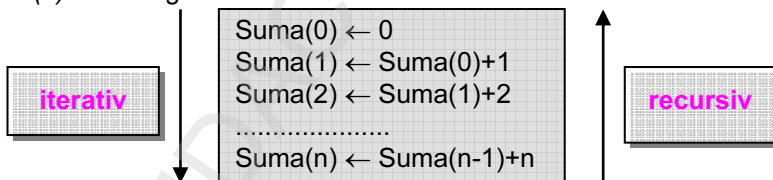
Scop: Exemplificarea modului de descompunere a problemei în procese recursive.

Enunțul problemei. Să se calculeze suma primelor n numere naturale. Valoarea lui n se introduce de la tastatură.

Această problemă poate fi rezolvată prin doi algoritmi:

- folosind **iterația** (algoritm **iterativ**);
- folosind **recursivitatea** (algoritm **recursiv**).

Acești algoritmi (iterativ sau recursiv) se bazează pe următoarea metodă – se consideră funcția $\text{suma}(n)$ care se generează astfel:



Acestui proces de calcul i se pot asocia două **funcții matematice**:

- funcția **iterativă** – definiția iterativă a unei funcții se face printr-o expresie care conține numai valori cunoscute;
- funcția **recursivă** – definiția recursivă a unei funcții se face printr-o expresie care conține însăși funcția.

Funcția matematică iterativă asociată acestui proces de calcul este cea alăturată, iar programul care folosește algoritmul iterativ este:

$$\text{Suma}(n) = \begin{cases} 0 & \text{pentru } n=0 \\ 1+2+3+\dots+n & \text{pentru } n \neq 0 \end{cases}$$

```
#include <iostream.h>
int suma(int n)
{int i,s=0;
if (n==0) return 0;
else {for (i=1;i<=n;i++) s=s+i; return s;}}
```

```
void main()
{int n; cout<<"n= "; cin>>n; cout<<"suma= "<<suma(n);}
```

Funcția matematică recursivă asociată acestui proces de calcul este cea alăturată, iar programul care folosește algoritmul recursiv este:

$$\text{Suma}(n) = \begin{cases} 0 & \text{pentru } n=0 \\ n+\text{Suma}(n-1) & \text{pentru } n \neq 0 \end{cases}$$

```
#include <iostream.h>
int suma(int n)
{if (n==0) return 0; else return suma(n-1)+n;}
void main()
{int n; cout<<"n= "; cin>>n; cout<<"suma= "<<suma(n);}
```

Observații:

1. Recurența este realizată prin **autoapelul funcției suma**.
2. În algoritmul recursiv, pentru calcularea sumei sunt necesare două elemente:
 - **formula de recurență**: $\text{suma}(n) = n + \text{suma}(n-1)$
 - **o valoare inițială cunoscută**: $\text{suma}(0) = 0$
3. În algoritmul recursiv, numele funcției poate să apară în corpul funcției și în membrul stâng al unei instrucțiuni de atribuire, spre deosebire de algoritmul iterativ, unde poate să apară numai în membrul drept al instrucțiunii de atribuire. Din această cauză, în algoritmul iterativ se folosește o variabilă suplimentară **s** pentru calculul sumei.
4. Ideea de bază a recursivității este aceea că fiecare nou autoapel al funcției (autogenerarea unui nou proces de același fel) ne apropie de soluția finală, care corespunde valorii inițiale, cunoscute.



Studiu de caz

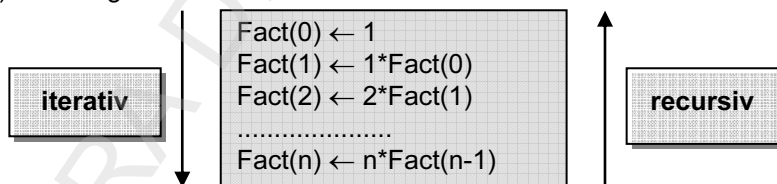
Scop: Exemplificarea modului în care se execută apelurile recursive.

Enunțul problemei. Să se calculeze factorialul unui număr n : $n! = 1 \times 2 \times 3 \times \dots \times n$. Valoarea lui n se introduce de la tastatură.

Și această problemă poate fi rezolvată prin doi algoritmi:

- folosind **iterația** (algoritm **iterativ**);
- folosind **recursivitatea** (algoritm **recursiv**).

Algoritmul (iterativ sau recursiv) se bazează pe următoarea metodă – se consideră funcția $\text{fact}(n)$ care se generează astfel:



Funcția matematică iterativă asociată acestui proces de calcul este cea alăturată, iar programul care folosește algoritmul iterativ este:

$$\text{Fact}(n) = \begin{cases} 1 & \text{pentru } n=0 \\ 1 \times 2 \times 3 \times \dots \times (n-1) \times n & \text{pentru } n \neq 0 \end{cases}$$

```
#include <iostream.h>
unsigned long fact(int n)
{int i; unsigned long p=1;
```



```

if (n==0) return 1;
    else {for (i=1;i<=n;i++) p*=i; return p;}}
void main()
{int n; cout<<"n= "; cin>>n; cout<<"factorial= "<<fact(n);}

```

Funcția matematică recursivă asociată acestui proces de calcul este cea alăturată, iar programul care folosește algoritmul recursiv este:

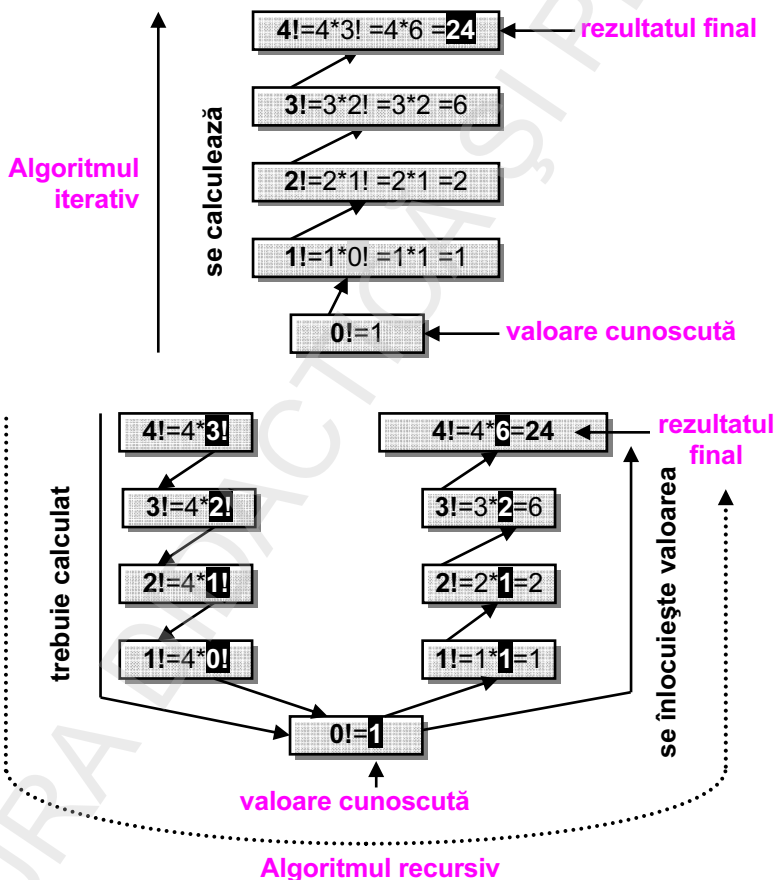
$$\text{Fact}(n) = \begin{cases} 1 & \text{pentru } n=0 \\ n \times \text{Fact}(n-1) & \text{pentru } n \neq 0 \end{cases}$$

```

#include <iostream.h>
unsigned long fact(int n)
{if (n==0) return 1; else return n*fact(n-1);}
void main()
{int n; cout<<"n= "; cin>>n; cout<<"factorial= "<<fact(n);}

```

Să analizăm modul în care se execută cei doi algoritmi, pentru a calcula valoarea funcției $\text{fact}(4)=4!$.



Se observă că:

1. Execuția algoritmului iterativ se face într-o singură etapă, care constă în rezolvarea problemei pornind de la o valoare inițială cunoscută, până la obținerea valorii finale (valoarea cerută), adică „de jos în sus“.

2. Execuția algoritmului recursiv se face în două etape: în prima etapă se descompune problema, pornind de la valoarea cerută până la valoarea inițială, cunoscută, adică „de sus în jos”, iar în a doua, se rezolvă problema pornind de la valoarea inițială cunoscută până la obținerea valorii cerute, adică „de jos în sus”.



1.2.2. Reguli pentru construirea unui subprogram recursiv

1. Corpul unui subprogram recursiv descrie algoritmul folosind două elemente:
 - **Cazul general al soluției.** Conține toate prelucrările necesare pentru a reduce dimensiunea problemei în vederea apropierei de rezultatul final. Cea mai importantă operație care se execută este **autoapelul**. În exemplul funcției **fact**, cazul general al soluției este format din instrucțiunea **return n*fact(n-1);** care conține autoapelul.
 - **Cazul de bază al soluției.** Conține o operație care rezolvă un caz special al problemei, fără a se folosi autoapelul. În exemplul funcției **fact**, cazul de bază al soluției este format din instrucțiunea **return 1;** care nu conține autoapel.
2. Recursivitatea este un proces repetitiv și este obligatoriu să existe o condiție de oprire a repetiției. Cele două cazuri se combină folosind o structură alternativă **if...else**. Condiția de oprire a recursivității este exprimată printr-o expresie logică ce trebuie să depindă de parametrii subprogramului sau de variabilele sale locale (în exemplul funcției **fact** expresia logică este $(n==0)$). **Această expresie logică este folosită pentru a face trecerea de la cazul general la cazul de bază al soluției.**
3. Orice definiție recursivă trebuie să asigure **condiția de consistență**, adică valoarea funcției trebuie să fie direct calculabilă sau calculabilă cu ajutorul unei valori direct calculabile. Aceasta înseamnă că modificarea parametrilor și/sau a variabilelor locale, de la o activare la alta, trebuie să asigure condiția de ieșire din recursivitate. În exemplul următor, funcția recursivă nu îndeplinește condiția de consistență, deoarece modificarea parametrului n de la o activare la alta ($n=n+1$) ne îndepărtează de condiția de ieșire din recursivitate ($n=0$):

```
int s(int n);
{if n==0 return 0; else return n+s(n+1);}
```

Observație:

Din exemplele prezentate se observă că, înainte de a construi algoritmul recursiv, trebuie definită funcția matematică recursivă. Ea ajută la identificarea cazului de bază al soluției și a cazului general al soluției.

Exemplul 1

Să se calculeze suma:

$$S = 1 + 3 + 5 + 7 + \dots + (2n-1)$$

Suma este egală cu suma calculată anterior, la care se adaugă numărul, numai dacă este par.

$$\text{Suma}(n) = \begin{cases} 0 & \text{pentru } n=0 \\ \text{Suma}(n-1) & \text{pentru } n \text{ par} \\ n+\text{Suma}(n-1) & \text{pentru } n \text{ impar} \end{cases}$$

Exemplul 2

Să se calculeze:

$$S = 1^2 - 2^2 + 3^2 - 4^2 + \dots + (-1)^{2n+1} \times n^2$$

Suma este egală cu suma calculată anterior, la care se adaugă pătratul numărului, dacă este un număr impar, sau se scade pătratul numărului, dacă numărul este par.

$$\text{Suma}(n) = \begin{cases} 0 & \text{pentru } n=0 \\ \text{Suma}(n-1)+n*n & \text{pentru } n \text{ impar} \\ \text{Suma}(n-1)-n*n & \text{pentru } n \text{ par} \end{cases}$$

Temă

1. Identificați, în fiecare exemplu de funcție matematică recursivă, cazul de bază și cazul general al soluției. Scrieți câte un program în care calculați valoarea sumelor definite mai sus. Folosiți în fiecare program, pentru calculul sumei, o funcție implementată iterativ și o funcție implementată recursiv. Afișați ambele rezultate obținute, prin evaluarea fiecărei funcții. Comparați rezultatele obținute.
2. Scrieți câte o funcție matematică recursivă pentru calculul fiecărei expresii de mai jos. Implementați aceste funcții matematice cu ajutorul subprogramelor recursive:

$$S = 1 \times 3 + 2 \times 5 + 3 \times 7 + 4 \times 9 + \dots + n \times (2n+1)$$

$$S = 1 \times 2 - 2 \times 4 + 3 \times 6 - 4 \times 8 + \dots + (-1)^{n-1} \times n \times (2n)$$

$$P = 2 \times 4 \times 6 \times 8 \times \dots \times 2n$$

$$P = 1 \times 4 \times 7 \times 10 \times \dots \times (3n-2)$$
3. Scrieți un subprogram recursiv care calculează produsul a două numere naturale, **a** și **b**, prin adunarea repetată a lui **a** de **b** ori ($a \times b = a + a + a + \dots + a$).

1.2.3. Variabilele locale și subprogramele recursive

O variabilă locală a unui subprogram este definită în interiorul subprogramului și poate fi folosită numai în cadrul său. În cazul subprogramelor recursive, fiecare autoapel al subprogramului înseamnă un nou proces, căruia i se rezervă o nouă zonă de memorie internă în care să se execute. Implicit, înseamnă și o nouă definire a variabilei locale, adică o nouă zonă de memorie internă alocată pentru variabila locală, numai pentru acel apel al subprogramului.

Așadar, **variabila locală** este unică pentru un apel al subprogramului. Dacă subprogramul este apelat recursiv de *n* ori, vor exista *n* imagini ale aceleiași variabile locale, câte una pentru fiecare apel. Fiecare imagine va avea un conținut diferit, corespunzător apelului.

La activarea fiecărui subprogram recursiv, vârful stivei sistemului urcă, și se salvează în stivă instanța subprogramului apelant. În acest mod, în stivă se vor regăsi toate instanțele activării subprogramului recursiv. Înainte de a se reveni dintr-o activare a unui subprogram recursiv, vârful stivei coboară și este adus la valoarea anterioară activării.

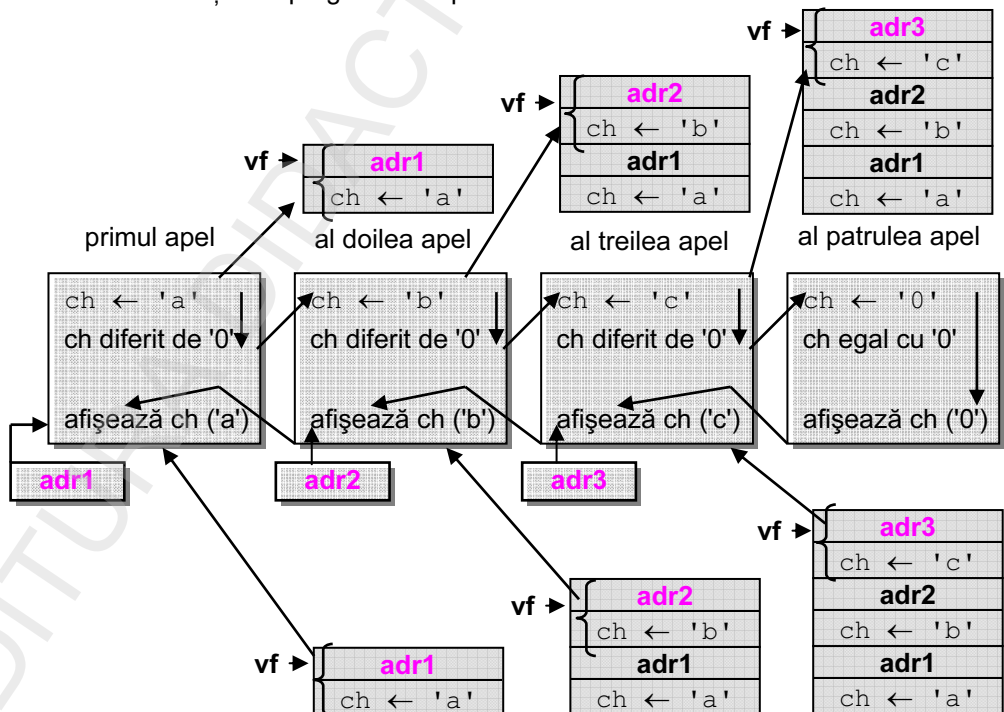
În exemplul următor este prezentat un subprogram recursiv în care se poate urmări acest comportament al variabilei locale (în exemplu, variabila **ch**). Se introduce un șir de caractere de la tastatură, caracter cu caracter, fără să se folosească un vector de caractere. Șirul de caractere se termină cu caracterul 0. Să se afișeze inversat acest șir de caractere.

```
#include <iostream.h>
void invsir()
{char ch; cin>>ch;
  if (ch!='0') invsir();
  cout<<ch;}
//if (ch!='0') cout<<ch;} - pentru a nu se afișa caracterul 0
void main()
{cout<<"Scrieti textul "; invsir();}
```

Să urmărim cum se execută acest program atunci când se introduce șirul de caractere "abc0":

1. Se apelează subprogramul **invsir()**.
2. Se creează prima imagine a variabilei locale **ch**, care corespunde primului apel.

3. Se citește valoarea variabilei **ch**: 'a'. Prima imagine a variabilei locale conține valoarea 'a'. Fiind diferită de '0', se apelează din nou subprogramul **invsir()**.
4. Se creează a doua imagine a variabilei locale **ch**, care corespunde celui de al doilea apel.
5. Se citește valoarea variabilei **ch**: 'b'. A doua imagine a variabilei locale conține valoarea 'b'. Fiind diferită de '0', se apelează din nou subprogramul **invsir()**.
6. Se creează a treia imagine a variabilei locale **ch**, care corespunde celui de al treilea apel.
7. Se citește valoarea variabilei **ch**: 'c'. A treia imagine a variabilei locale conține valoarea 'c'. Fiind diferită de '0', se apelează din nou subprogramul **invsir()**.
8. Se creează a patra imagine a variabilei locale **ch**, care corespunde celui de al patrulea apel.
9. Se citește valoarea variabilei **ch**: '0'. A patra imagine a variabilei locale conține valoarea '0'. Fiind '0', se trece la execuția instrucțiunii următoare din subprogram.
10. Se execută instrucțiunea **cout<<ch**; , prin care se scrie pe ecran valoarea păstrată în imaginea variabilei **ch**, corespunzătoare celui de al patrulea apel – '0'.
11. Se termină de executat al patrulea apel al subprogramului. Se predă controlul celui de al treilea subprogram apelat. Instrucțiunea care urmează să se execute este **cout<<ch**; , prin care se afișează pe ecran valoarea păstrată în imaginea variabilei **ch**. Conținutul corespunzător celui de al treilea apel este 'c'.
12. Se termină de executat al treilea apel al subprogramului. Se predă controlul celui de al doilea subprogram apelat. Instrucțiunea care urmează să se execute este **cout<<ch**; , prin care se afișează pe ecran valoarea păstrată în imaginea variabilei **ch**. Conținutul corespunzător celui de al doilea apel este 'b'.
13. Se termină de executat al doilea apel al subprogramului. Se predă controlul primului subprogram apelat. Instrucțiunea care urmează să se execute este **cout<<ch**; , prin care se afișează pe ecran valoarea păstrată în imaginea variabilei **ch**. Conținutul corespunzător primului apel este 'a'.
14. Se termină execuția subprogramelor apelate.



Adâncimea recursivității este numărul de activări (apeluri) succesive ale unui subprogram recursiv.

Pentru exemplul anterior, adâncimea recursivității este 4 (au fost patru apeluri).

Observație: Implementarea recursivă a unui algoritm este eficientă numai dacă adâncimea recursivității nu este mare.



Scrieți un program prin care se citesc n numere întregi de la tastatură (n este o valoare care se citește de la tastatură) și se afișează în ordinea inversă citirii. Nu se va folosi un vector pentru numere.

1.2.4. Implementarea recursivă a algoritmilor elementari

1.2.4.1. Algoritmul pentru determinarea valorii minime (maxime)

Pentru calcularea valorii minime dintr-un șir de n numere citite de la tastatură, se vor folosi funcțiile `min()` și `max()` care au fost definite recursiv:

```
#include <iostream.h>
int max(int a,int b){if (a>b) return a; else return max(b,a);}
int min(int a,int b){if (a>b) return b; else return min(b,a);}
void main()
{int a,b,x,y,n,i;
cout<<"n= "; cin>>n; cout<<"numar= "; cin>>a; x=a; y=a;
for (i=2;i<=n;i++)
    {cout<<"numar= "; cin>>a; x=max(x,a); y=min(x,a);}
cout<<"maxim= "<<x<<endl<<"minim= "<<y;}
```

1.2.4.2. Algoritmul pentru calculul c.m.m.d.c. a două numere

Un alt algoritm elementar pe care îl putem implementa folosind recursivitatea este calculul celui mai mare divizor comun dintre două numere naturale a și b . Valorile lui a și b se introduc de la tastatură.

Varianța 1. Soluția recursivă se bazează pe **algoritmul lui Euclid** (condiția $b \neq 0$):

1. Se împarte a la b și se obține restul r ($r \leftarrow a \bmod b$).
2. Se execută operațiile de atribuire $a \leftarrow b$; $b \leftarrow r$.
3. Dacă $b \neq 0$, atunci se revine la pasul 1, adică se reia execuția funcției atribuindu-se parametrilor de intrare următoarele valori: parametrului a valoarea b , iar parametrului b valoarea $a \bmod b$. Dacă $b = 0$, atunci $\text{cmmdc} \leftarrow a$.

Pentru algoritmul recursiv se definește funcția matematică recursivă:

$$\text{cmmdc}(a,b) = \begin{cases} a & \text{pentru } b=0 \\ b & \text{pentru } a=0 \\ \text{cmmdc}(b, a \bmod b) & \text{pentru } a \neq 0 \text{ și } b \neq 0 \end{cases}$$

De exemplu, dacă $a=18$ și $b=12$, calculul celui mai mare divizor comun se desfășoară, folosind algoritmul recursiv, astfel:

1. $\text{cmmdc}(18,12)$: $b \neq 0$ ($12 \neq 0$) și $a \neq 0$ ($18 \neq 0$) $\Rightarrow$
 $\text{cmmdc}(18,12) = \text{cmmdc}(12, 18 \bmod 12) = \text{cmmdc}(12,6)$.
2. $\text{cmmdc}(12,6)$: $b \neq 0$ ($6 \neq 0$) și $a \neq 0$ ($12 \neq 0$) $\Rightarrow$ $\text{cmmdc}(12,6) = \text{cmmdc}(6, 12 \bmod 6) = \text{cmmdc}(6,0)$.
3. $\text{cmmdc}(6,0)$: $b=0$ ($0=0$) $\Rightarrow$ $\text{cmmdc}(6,0) = a = 6$.

Programul care folosește algoritmul recursiv este:

```
#include <iostream.h>
int cmmdc(int a, int b)
{if (a==0) return b;
  else if (b==0) return a;
  else return cmmdc(b,a%b);}

void main()
{int a,b; cout<<"a= "; cin>>a; cout<<"b= "; cin>>b;
  if (cmmdc(a,b)==0)
    cout<<"cmmdc nu se poate calcula; ambele numere sunt 0";
  else cout<<"cmmdc ("<<a<<" , "<<b<<" ) = "<<cmmdc(a,b); }
```

Varianța 2. Soluția recursivă se bazează pe **algoritmul prin scăderi repetate** (condiția $a \neq b$):

1. Se scade, din numărul mai mare, celălalt număr: **dacă** $a > b$, se execută operația de atribuire $a \leftarrow a - b$, iar **dacă** $b > a$ se execută operația de atribuire $b \leftarrow b - a$.
2. **Dacă** $a \neq b$, atunci se revine la pasul 1 adică se reia execuția funcției atribuindu-se parametrilor de intrare următoarele valori: **dacă** $a > b$ parametrului a i se atribuie valoarea $a - b$, iar **dacă** $b > a$ parametrului b i se atribuie valoarea $b - a$; în ambele cazuri, celălalt parametru rămâne neschimbat.

Dacă $a = b$, atunci $\text{cmmdc} \leftarrow a$.

Pentru algoritmul recursiv se definește funcția recursivă alăturată.

$$\text{cmmdc}(a,b) = \begin{cases} a & \text{pentru } b=0 \text{ sau } a=b \\ b & \text{pentru } a=0 \\ \text{cmmdc}(a-b, b) & \text{pentru } a>b \\ \text{cmmdc}(a, b-a) & \text{pentru } b>a \end{cases}$$

De exemplu, dacă $a=18$ și $b=12$,

calculul celui mai mare divizor comun se desfășoară folosind algoritmul recursiv, astfel:

1. **cmmdc(18,12):** ($b \neq 0$ ($12 \neq 0$) și $a \neq b$ ($18 \neq 12$)) și $a \neq 0$ ($18 \neq 0$) și $a > b$ ($18 > 12$) $\Rightarrow$ $\text{cmmdc}(18,12) = \text{cmmdc}(18-12,12) = \text{cmmdc}(6,12)$.
2. **cmmdc(6,12):** ($b \neq 0$ ($12 \neq 0$) și $a \neq b$ ($6 \neq 12$)) și $a \neq 0$ ($6 \neq 0$) și $b > a$ ($12 > 6$) $\Rightarrow$ $\text{cmmdc}(6,12) = \text{cmmdc}(6, 12-6) = \text{cmmdc}(6,6)$.
3. **cmmdc(6,6):** $a=b$ ($6=6$) $\Rightarrow \text{cmmdc}(6,6) = a = 6$.

Programul care folosește algoritmul recursiv este:

```
#include <iostream.h>
int cmmdc(int a, int b)
{if (a==0 || a==b) return b;
  else if (b==0) return a;
  else if (a>b) return cmmdc(a-b,b);
  else return cmmdc(a,b-a);}

void main()
{int a,b; cout<<"a= "; cin>>a; cout<<"b= "; cin>>b;
  if (cmmdc(a,b)==0)
    cout<<"cmmdc nu se poate calcula; ambele numere sunt 0";
  else cout<<"cmmdc ("<<a<<" , "<<b<<" ) = "<<cmmdc(a,b); }
```

1.2.4.3. Algoritmi pentru prelucrarea cifrelor unui număr

Exemplul 1 – Extragerea cifrelor unui număr. Ca exemplu de implementare recursivă a algoritmului care extrage cifrele unui număr, vom calcula suma cifrelor unui număr natural n . Algoritmul constă în extragerea repetată a unei cifre (calculând restul împărțirii la 10), din

ceea ce mai rămâne din număr după extragerea unei cifre (calculând câtul împărțirii la 10). Pașii care se execută sunt:

1. Se inițializează suma cu valoarea 0.
2. Dacă numărul este diferit de 0, se adună la sumă ultima cifră din număr ($n \bmod 10$) și se elimină această cifră din număr ($n \div 10$) după care se reia pasul 2, adică se reia execuția funcției atribuind parametrului de intrare valoarea $n \div 10$. Dacă numărul n are valoarea 0, atunci s-a obținut suma cifrelor.

Pentru algoritmul recursiv, se definește funcția alăturată:

$$\text{Suma}(n) = \begin{cases} 0 & \text{pentru } n=0 \\ n \bmod 10 + \text{Suma}(n \div 10) & \text{pentru } n>0 \end{cases}$$

De exemplu, dacă se calculează suma cifrelor numărului 5372, funcția recursivă se va executa astfel:

$\text{Suma}(5372) = 5372 \bmod 10 + \text{Suma}(5372 \div 10) = 2 + \text{Suma}(537) = 2 + (537 \bmod 10 + \text{Suma}(537 \div 10)) = 2 + 7 + \text{Suma}(53) = 9 + (53 \bmod 10 + \text{Suma}(53 \div 10)) = 9 + 3 + \text{Suma}(5) = 12 + 5 \bmod 10 + \text{Suma}(5 \div 10) = 12 + 5 + \text{Suma}(0) = 17 + 0 = 17$

```
#include <iostream.h>
int suma(int n)
{if (n==0) return 0; else return n%10+suma(n/10);}
void main()
{int n; cout<<"n= "; cin>>n; cout<<"suma cifrelor= "<<suma(n);}
```

Temă



Scriveți o funcție recursivă prin care calculați suma cifrelor pare dintr-un număr.

Exemplul 2 – Compunerea unui număr cunoscând cifrele sale. Cifrele se citesc de la tastatură prin intermediul variabilei de memorie **cifra**. Funcția are un singur parametru: **n**, care reprezintă valoarea numărului obținut din cifrele obținute până la apel. Inițial, valoarea parametrului **n** (valoarea la primul apel) este 0. Dacă numărul citit este o cifră, se reia execuția funcției atribuindu-se parametrilor de intrare vechea valoare înmulțită cu 10, la care se adaugă cifra citită. Dacă numărul citit nu este o cifră, atunci funcția va returna valoarea finală a numărului **n**.

$$\text{Numar}(n) = \begin{cases} n & \text{pentru } \text{cifra} < 0 \text{ sau } \text{cifra} > 9 \\ \text{Numar}(n*10+\text{cifra}) & \text{pentru } \text{cifra} \geq 0 \text{ și } \text{cifra} \leq 9 \end{cases}$$

Implementarea recursivă cu ajutorul unui subprogram de tip funcție operand – se evaluează funcția **numar(0)**:

```
#include <iostream.h>
unsigned long numar(unsigned long n)
{int cif; cout<<"cifra= "; cin>>cif;
if (cif<0 || cif>9) return n; else return numar(n*10+cif);}
void main() {cout<<"numarul= "<<numar(0);}
```

și implementarea recursivă, cu ajutorul unui subprogram de tip funcție procedurală:

```
#include <iostream.h>
void numar(int &n)
{int cif; cout<<"cifra= "; cin>>cif;
```



```
if (cif>=0 && cif<=9) {n=n*10+cif; numar(n);}}
void main()
{int n=0; numar(n); cout<<"numarul= "<<n; }
```

Exemplul 3 – Obținerea inversului unui număr. Pornind de la variantele recursive definite anterior, pentru extragerea cifrelor unui număr și pentru compunerea unui număr din cifrele sale, inversul unui număr n se poate determina folosind un algoritm recursiv în care nr reprezintă valoarea numărului inversat. Inițial, valoarea parametrului nr (la primul apel) este 0.

$$\text{Inv}(n,nr) = \begin{cases} n+nr & \text{pentru } n < 10 \\ \text{Inv}(n \text{ div } 10, 10 * (nr + (n \bmod 10))) & \text{pentru } n \geq 10 \end{cases}$$

Implementarea cu ajutorul unui subprogram de tip funcție operand – se evaluează funcția $\text{inv}(n, 0)$:

```
#include <iostream.h>
unsigned long inv(unsigned long n, unsigned long nr)
{if (n<10) return nr+n; else return inv(n/10, 10*(nr+n%10));}
void main()
{unsigned long n;
cout<<"n= "; cin>>n; cout<<"invers= "<<inv(n,0); }
```

și implementarea recursivă, cu ajutorul unui subprogram de tip funcție procedurală:

```
#include <iostream.h>
void inv(int n,int &nr)
{if(n<10) nr=nr+n; else {nr=10*(nr+n%10); inv(n/10,nr);}}
void main()
{int n,nr=0; cout<<"n= "; cin>>n; inv(n,nr); cout<<nr<<endl;}
```

Temă



Folosind funcțiile recursive definite anterior, scrieți un program prin care afișați numerele palindrom din intervalul $[a, b]$ – a și b se citesc de la tastatură.

1.2.4.4. Algoritmul pentru verificarea unui număr dacă este prim

Pentru a determina dacă un număr natural n este prim, se poate defini funcția iterativă $\text{prim}(n, i)$, astfel:

$$\text{prim}(n, i) = \begin{cases} \text{true} & \text{dacă } n \text{ nu este divizibil prin } i, \forall i \in [2, \sqrt{n}] \\ \text{false} & \text{dacă } n \text{ este divizibil prin } i, i \in [2, \sqrt{n}] \end{cases}$$

Din algoritmul iterativ se observă că funcția **prim** are valoarea **true** numai dacă numărul n nu este divizibil cu nici unul dintre numerele cuprinse între 2 și $[\sqrt{n}]$. Altfel spus, se evaluează toate funcțiile $\text{prim}(n, i)$, și dacă cel puțin una dintre aceste funcții are valoarea **false**, atunci numărul nu este prim. Algoritmul recursiv poate fi descris prin funcția:

$$\text{prim}(n, i) = \begin{cases} \text{true} & \text{pentru } i=1 \\ (n \bmod i \neq 0) \text{ and } \text{prim}(n, i-1) & \text{pentru } 2 \leq i \leq [\sqrt{n}] \end{cases}$$

și se evaluează funcția $\text{prim}(n, \sqrt{n})$:

```
#include <iostream.h>
#include <math.h>
int prim(int n,int i)
{if(i==1) return 1;
  else return (n%i!=0) && prim(n,i-1);}
void main()
{int n; cout<<"n= "; cin>>n;
  if (prim(n,sqrt(n))) cout<<"Numarul "<<n<<" este prim ";
  else cout<<"Numarul "<<n<<" nu este prim ";
```

sau prin funcția:

$$\text{prim}(n,i) = \begin{cases} \text{true} & \text{pentru } i=[\text{sqrt}(n)] \\ (n \bmod i \neq 0) \text{ and } \text{prim}(n,i+1) & \text{pentru } 1 \leq i < [\text{sqrt}(n)] \end{cases}$$

și se evaluează funcția **prim(n,2)**:

```
#include <iostream.h>
#include <math.h>
int prim(int n,int i)
{if (i==sqrt(n)) return 1;
  else return (n%i!=0) && prim(n,i+1);}
void main()
{int n; cout<<"n= "; cin>>x;
  if (prim(n,2)) cout<<"Numarul "<<n<<" este prim ";
  else cout<<"Numarul "<<n<<" nu este prim ";
```

În cazul celor două variante definite pentru funcția recursivă se fac multe autoapeluri inutile. De aceea, se poate îmbunătăți funcția recursivă, definindu-se în felul următor:

$$\text{prim}(n,i) = \begin{cases} \text{true} & \text{pentru } i=1 \\ \text{false} & \text{pentru } n \bmod i = 0 \text{ și } i \neq 1 \\ \text{prim}(n,i-1) & \text{pentru } n \bmod i \neq 0 \text{ și } i \neq 1 \end{cases}$$

```
#include <iostream.h>
#include <math.h>
int prim(int n,int i)
{if(i==1) return 1;
  else if (n%i==0) return 0;
  else return prim(n,i-1);}
void main()
{int n; cout<<"n= "; cin>>x;
  if (prim(n,sqrt(n))) cout<<"Numarul "<<n<<" este prim ";
  else cout<<"Numarul "<<n<<" nu este prim ";
```

Temă



Arătați cum se execută funcția recursivă **prim()** pentru a testa numărul 27 și pentru a testa numărul 29. Refaceți funcția recursivă astfel încât, în programul principal, să evaluați funcția **prim(n,2)**. Folosiți funcția definită recursiv pentru a afișa primele **n** numere prime.

1.2.4.5. Algoritmi pentru determinarea divizorilor unui număr

Exemplul 1: Afișarea divizorilor unui număr. Se generează divizorii posibili (d) ai numărului n , pornind de la valoarea 1. Dacă divizorul posibil este mai mic decât $n/2$, se verifică dacă numărul n este divizibil prin el. Dacă este divizibil, se afișează divizorul d . Se reia apoi execuția funcției, pentru următorul divizor posibil. Se va evalua funcția **divizor**($n,1$).

```
#include <iostream.h>
void divizor(int n, int d)
{if (d<=n/2)
    {if (n%d==0) cout<<d<<" "; divizor(n,d+1);}}
void main()
{int n; cout<<"n= "; cin>>n; divizor(n,1);}
```

sau

```
#include <iostream.h>
void divizor(int n, int d)
{if (d<=sqrt(n))
    {if (n%d==0)
        if (d!=n/d) cout<<d<<" "<<n/d<<" ";
        else cout<<d<<" ";
        divizor(n,d+1);}}
void main()
{int n; cout<<"n= "; cin>>n; divizor(n,1);}
```

Exemplul 2: Să se descompună un număr în factori primi. Se generează factorii primi posibili (f) ai numărului n , pornind de la valoarea 2 a factorului, până când se elimină din număr toți factorii primi ($n=1$). Se folosește parametrul p pentru a calcula exponentul factorului prim. Dacă f este factor prim, se apelează funcția pentru numărul din care s-a eliminat un factor prim (n/f) și pentru valoarea exponentului p incrementată cu 1. Dacă f nu este factor prim în numărul n obținut, se verifică dacă a fost factor prim ($p \neq 0$) – caz în care se afișează factorul prim (f) și exponentul (p) – după care se apelează funcția pentru următorul factor prim posibil ($f+1$) cu exponentul 0. Dacă s-au eliminat din număr toți factorii primi, se verifică dacă ultimul factor a fost factor prim ($p \neq 0$) – caz în care se afișează factorul prim (f) și exponentul (p). Se va evalua funcția **factor**($n,2,0$).

```
#include <iostream.h>
void factor(int n, int f, int p)
{if (n>1)
    if (n%f==0) factor(n/f,f,p+1);
    else {if (p!=0) cout<<f<<" la puterea "<<p<<endl;
        factor(n,f+1,0);}
    else if (p!=0) cout<<f<<" la puterea "<<p;
void main()
{int n; cout<<"n= "; cin>>n; factor(n,2,0); }
```

Temă



Scrieți un program prin care afișați suma divizorilor primi ai unui număr. Folosiți o funcție definită recursiv, prin care determinați divizorul prim al numărului, după care îl adunați la sumă.

1.2.4.6. Algoritmi pentru conversia între baze de numerație

Se convertește un număr n scris în baza 10 într-un număr scris în baza q ($1 < q < 10$). Dacă $n < q$, înseamnă că numărul este reprezentat deja în baza q , deoarece se folosesc aceleași cifre pentru reprezentarea lui. În cazul în care $n \geq q$ algoritmul de conversie este:

1. Se atribuie câtului valoarea numărului $c \leftarrow n$.
2. Se calculează restul împărțirii numărului la valoarea q ($r \leftarrow n \bmod q$) – o cifră a numărului – și se afișează (primul rest este cifra cea mai puțin semnificativă).
3. Se evaluează c – câtul împărțirii numărului la q . Dacă $c \geq q$, se revine la pasul 2, adică se reia execuția subprogramului, atribuind parametrului de intrare valoarea $n \div q$. Dacă $c < q$, se termină procesul repetitiv și câtul obținut va fi ultima cifră (cifra cea mai semnificativă).

Tipărirea cifrelor obținute se face în ordinea inversă a extragerii (la fel ca și în cazul funcției recursive, prin care se afișează inversul unui șir de caractere). Înseamnă că, la fiecare apel al funcției, cifra obținută prin calcularea restului se va păstra de fiecare dată în câte o imagine a variabilei de memorie folosită în funcția recursivă – variabila **cifra**. Tipărirea cifrelor se va face în ordinea inversă a obținerii lor, deoarece prima cifră care se va tipări va fi cea corespunzătoare ultimului apel, iar ultima cifră care se va tipări va fi cea corespunzătoare primului apel:

```
#include <iostream.h>
void conversie(int n, int q)
{int cifra; cifra=n%q;
 if (n>=q) conversie(n/q,q);
 cout<<cifra;}
void main()
{int n,q; cout<<"numarul= "; cin>>n; cout<<"baza= "; cin>>q;
 conversie(n,q);}
```

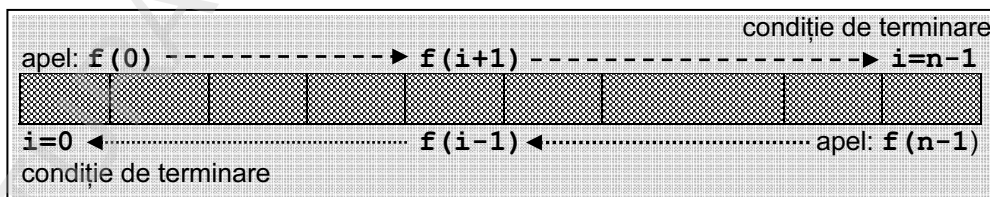
Temă



Arătați cum se execută funcția recursivă **conversie(25,3)**. Scrieți o funcție recursivă prin care converțiți în baza 10 un număr b scris în baza q ($1 < q < 10$).

1.2.5. Implementarea recursivă a algoritmilor pentru prelucrarea tablourilor de memorie

Prelucrarea unui vector printr-o funcție recursivă f se poate face începând cu primul element ($i=0$) și terminând cu ultimul element ($i=n-1$), sau invers, începând cu ultimul element ($i=n-1$) și terminând cu primul element ($i=0$):



Exemplul 1

Se citesc și se afișează elementele întregi ale unui vector. Cele două funcții recursive: pentru citirea valorii elementelor de la tastatură – `citeste()` – și pentru afișarea valorii elementelor vectorului – `scrie()` – pot începe cu primul element din vector și se termină cu ultimul element:

```
#include <iostream.h>
int n,a[100];
void citeste(int i)
{if(i!=n-1) citeste(i+1);
 cout<<i<<": "; cin>>a[i];}
void scrie(int i)
{if(i!=n-1) scrie(i+1);
 cout<<a[i]<<" ";}
void main()
{cout<<"n= "; cin>>n; citeste(0); scrie(0);}
```

sau pot începe cu ultimul element din vector și se termină cu primul element:

```
#include <iostream.h>
int n,a[100];
void citeste(int i)
{if(i!=0) citeste(i-1);
 cout<<i<<": "; cin>>a[i];}
void scrie(int i)
{if(i!=0) scrie(i-1);
 cout<<a[i]<<" ";}
void main()
{cout<<"n= "; cin>>n; citeste(n-1); scrie(n-1);}
```

Exemplul 2

Să se verifice dacă există, într-un vector cu n elemente întregi, cel puțin un element cu valoarea întreagă x . Fie vectorul $a=(a_1, a_2, \dots, a_n)$. Funcția iterativă

$$\text{gasit}(x,i) = \begin{cases} \text{true} & \text{dacă } x \in (a_1, a_2, \dots, a_n) \\ \text{false} & \text{dacă } x \notin (a_1, a_2, \dots, a_n) \end{cases}$$

$\text{gasit}(x,i)$ este definită alăturat. Din algoritmul iterativ se observă că funcția gasit are valoarea **true** fie dacă elementul curent are valoarea x , fie dacă cel puțin unul dintre elementele anterior testate are valoarea x . Funcția recursivă poate fi definită în mai multe moduri.

Varianta 1 – se evaluează funcția $\text{gasit}(x, n-1)$:

```
#include <iostream.h>
int a[100];
int gasit(int x,int i)
{if(i==0) return x==a[0];
 else return ((x==a[i]) || gasit(x, i-1));}
void main()
{int x,i,n; cout<<"n= "; cin>>n; cout<<"x= "; cin>>x;
 for(i=0;i<n;i++) {cout<<"a["<<i<<"]=" "; cin>>a[i];}
 if(gasit(x,n-1)) cout<<"s-a gasit elementul"<<x;
 else cout<<"nu s-a gasit elementul"<<x;}
```

$$\text{gasit}(x,i) = \begin{cases} x=a_0 & \text{pentru } i=0 \\ (x=a_i) \text{ or } \text{gasit}(x,i-1) & \text{pentru } 1 \leq i \leq n-1 \end{cases}$$

De exemplu, pentru $n=5$, $x=2$ și $a=(1,1,2,2,3)$, evaluarea funcției se face astfel:

$\text{gasit}(2,4) = (2=3) \text{ or } \text{gasit}(2,3) = \text{false or } ((2=2) \text{ or } \text{gasit}(2,2)) = \text{false or true or } ((2=2) \text{ or } \text{gasit}(2,1)) = \text{false or true or true or } (2=1) \text{ or } \text{gasit}(2,0)) = \text{false or true or true or false or } (2=1) = \text{false or true or true or false or false} = \text{true}$

Varianta 2 – se evaluează funcția $\text{gasit}(x, 0)$:

```
#include <iostream.h>
int a[100];
int gasit(int x,int i)
{if(i==n-1) return x==a[n-1];
  else return (x==a[i]) || gasit(x,i+1);}

void main()
{int x,i,n;
  cout<<"n= "; cin>>n;
  for(i=0;i<n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
  cout<<"x= "; cin>>x;
  if(gasit(x,0)) cout<<"s-a gasit elementul"<<x;
  else cout<<"nu s-a gasit elementul"<<x;}
```

$$\text{gasit}(x,i) = \begin{cases} x=a_{n-1} & \text{pentru } i=n-1 \\ (x=a_i) \text{ or } \text{gasit}(x,i+1) & \text{pentru } 0 \leq i \leq n-2 \end{cases}$$

Varianta 3 – Din evaluarea funcției $\text{gasit}(2,4)$ se observă că valoarea funcției s-a aflat după al doilea apel al funcției și restul apelurilor au fost inutile. Înseamnă că, imediat ce s-a găsit un element cu valoarea x , funcția nu trebuie să se mai autoapeleze, și se atribuie funcției valoarea logică *true*. Pentru a evita însă apelarea la infinit a funcției în cazul în care nici un element din vector nu are valoarea x , se va folosi o condiție suplimentară ($i \geq n$). Funcția recursivă este definită alăturat și se va evalua funcția $\text{gasit}(x, 0)$:

```
#include <iostream.h>
int a[100];
int gasit(int x,int i)
{if(i>=n) return 0;
  else if (x==a[i]) return 1;
  else return gasit(x,i+1);}

void main()
{int x,i,n; cout<<"n= "; cin>>n; cout<<"x= "; cin>>x;
  for(i=0;i<n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
  if(gasit(x,0)) cout<<"s-a gasit elementul"<<x;
  else cout<<"nu s-a gasit elementul"<<x;}
```

$$\text{gasit}(x,i) = \begin{cases} \text{false} & \text{pentru } i \geq n \\ \text{true} & \text{pentru } x=a_i \text{ și } i < n \\ \text{gasit}(x,i+1) & \text{pentru } x \neq a_i \text{ și } i < n \end{cases}$$

Exemplul 3

Să se verifice dacă există, într-un vector cu n elemente întregi, cel puțin un element pozitiv. Fie vectorul $a=(a_1, a_2, \dots, a_n)$. Funcția iterativă $\text{pozitiv}(i)$ este definită alăturat. Din

definiția iterativă se observă că funcția pozitiv are valoarea *true* fie dacă elementul curent este pozitiv, fie dacă cel puțin unul dintre elementele anterior testate este pozitiv. Altfel spus, se evaluează toate funcțiile $\text{pozitiv}(i)$, și dacă cel puțin una dintre aceste funcții are valoarea *true*, atunci există cel puțin un element cu valoare pozitivă. Funcția recursivă poate fi definită în următoarele variante.

$$\text{pozitiv}(i) = \begin{cases} \text{true } a_i > 0 \\ \text{false } a_i \leq 0 \end{cases}$$

Varianta 1 – se evaluează funcția **pozitiv(n-1)**:

```
#include <iostream.h>
```

```
int a[100],n;
```

```
int pozitiv(int i)
```

```
{if(i==0) return a[0]>0;
```

```
    else return (a[i]>0) || pozitiv(i-1);}
```

```
void main()
```

```
{int i; cout<<"n= "; cin>>n;
```

```
    for(i=0;i<n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
```

```
    if(pozitiv(n-1)) cout<<"S-a gasit cel puțin un element pozitiv";
```

```
    else cout<<"Nu s-a gasit nici un element pozitiv";}
```

$$\text{pozitiv}(i) = \begin{cases} a_0 > 0 & \text{pentru } i=0 \\ (a_i > 0) \text{ or } \text{pozitiv}(i-1) & \text{pentru } 1 \leq i \leq n-1 \end{cases}$$

Varianta 2 – se evaluează funcția **pozitiv(0)**:

```
#include <iostream.h>
```

```
int a[100],n;
```

```
int pozitiv(int i)
```

```
{if(i==n-1) return a[n-1]>0;
```

```
    else return ((a[i]>0) || pozitiv(i+1));}
```

```
void main()
```

```
{int i; cout<<"n= "; cin>>n;
```

```
    for(i=0;i<n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
```

```
    if(pozitiv(0)) cout<<"S-a gasit cel puțin un element pozitiv";
```

```
    else cout<<"Nu s-a gasit nici un element pozitiv";}
```

$$\text{pozitiv}(i) = \begin{cases} a_{n-1} > 0 & \text{pentru } i=n-1 \\ (a_i > 0) \text{ or } \text{pozitiv}(i+1) & \text{pentru } 0 \leq i \leq n-2 \end{cases}$$

Varianta 3 – Și în cazul celor două variante definite anterior pentru funcția recursivă se vor face multe autoapeluri inutile. Ar trebui ca, imediat ce s-a găsit un element pozitiv, să nu se mai apeleze recursiv funcția și să i se atribue valoarea *true*. Pentru a evita însă apelarea la infinit a funcției, și în acest caz se va folosi o condiție suplimentară ($i <= n$). Funcția recursivă este definită alăturat. Se evaluează funcția **pozitiv(0)**

$$\text{pozitiv}(i) = \begin{cases} \text{false} & \text{pentru } i > n \\ \text{true} & \text{pentru } a_i > 0 \\ \text{pozitiv}(i+1) & \text{pentru } a_i \leq 0 \end{cases}$$

```
#include <iostream.h>
```

```
int a[100],n;
```

```
int pozitiv(int i)
```

```
{if(i>=n) return 0;
```

```
    else if (a[i]>0) return 1;
```

```
    else return pozitiv(i+1);}
```

```
void main()
```

```
{int i; cout<<"n= "; cin>>n;
```

```
    for(i=0;i<n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
```

```
    if(pozitiv(0)) cout<<"S-a gasit cel puțin un element pozitiv";
```

```
    else cout<<"Nu s-a gasit nici un element pozitiv";}
```

Exemplul 4

Să se calculeze numărul de apariții ale unui element precizat, **x**, într-un vector cu **n** elemente. Fie vectorul **a=(a₁, a₂, ..., a_n)**. Funcția iterativă **gasit(x, i)** este definită alăturat.

$$\text{gasit}(x, i) = \begin{cases} \text{gasit}(x, i-1) & \text{dacă } a_i \neq x \\ \text{gasit}(x, i-1) + 1 & \text{dacă } a_i = x \end{cases}$$

Din algoritmul iterativ se observă că funcției **gasit** i se incrementează valoarea dacă elementul curent are valoarea **x**. Funcției i se va atribui valoarea 0 dacă primul element comparat nu are valoarea **x**, sau valoarea 1 dacă primul element comparat are valoarea **x**. Mai cunoașteți că, în limbajul C++, valorii logice **false** îi corespunde valoarea numerică 0, iar valorii logice **true**, valoarea numerică 1. Notăm cu **num** ($x=a_i$) valoarea numerică asociată valorii logice a operatorului relațional. Funcția recursivă poate fi definită în două variante, în funcție de elementul din vector care se compară primul cu valoarea **x**.

Varianta 1 – se evaluează funcția **gasit(x,n-1)**.

```
#include <iostream.h>
int a[100],n;
int gasit(int x,int i)
{if (i==0) return x==a[0];
  else return (x==a[i])+gasit(x,i-1);}
void main()
{int i,x; cout<<"n= "; cin>>n; cout<<"x= "; cin>>x;
  for (i=0;i<n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
  cout<<"S-au gasit "<<gasit(x,n)<<"elemente"; }
```

$$\text{gasit}(x,i) = \begin{cases} \text{num}(x=a_0) & \text{pentru } i=0 \\ \text{num}(x=a_i) + \text{gasit}(x,i-1) & \text{pentru } 1 \leq i \leq n-1 \end{cases}$$

Varianta 2 – se evaluează funcția **gasit(x,0)**.

```
#include <iostream.h>
int a[100],n;
int gasit(int x,int i)
{if (i==n-1) return x==a[n-1];
  else return (x==a[i])+gasit(x,i+1);}
void main()
{int i,x; cout<<"n= "; cin>>n; cout<<"x= "; cin>>x;
  for (i=1;i<=n;i++) {cout<<"a["<<i<<"]= "; cin>>a[i];}
  cout<<"S-au gasit "<<gasit(x,0)<<"elemente"; }
```

$$\text{gasit}(x,i) = \begin{cases} \text{num}(x=a_{n-1}) & \text{pentru } i=n-1 \\ \text{num}(x=a_i) + \text{gasit}(x,i-1) & \text{pentru } 0 \leq i \leq n-2 \end{cases}$$

Exemplul 5

Să se caute, într-un vector ordonat, cu **n** elemente întregi, elementul cu valoarea întreagă **x**, și să se afișeze poziția în care a fost găsit. Dacă nu se găsește elementul, funcția va furniza valoarea -1. Fie vectorul **a=(a₁,a₂,...,a_n)**. Pentru căutarea elementului cu valoarea precizată se va folosi algoritmul de căutare binară, unde **s** și **d** sunt indicele din stânga, respectiv din dreapta, ai subvectorului în care se continuă căutarea, iar **m** este indicele elementului din mijloc: $m=(s+d)/2$. Funcția recursivă **gasit(s,d,a,x)** este definită alăturat. Se evaluează funcția **gasit(0,n-1,a,x)**.

```
#include <iostream.h>
int gasit(int s,int d,int a[],int x)
{int m=(s+d)/2;
  if(s>d) return -1;
  else if (a[m]==x) return m+1;
  else if (a[m]>x) return gasit(s,m-1,a,x);
  else return gasit(m+1,d,a,x); }
```

$$\text{gasit}(s,d,x,a) = \begin{cases} -1 & \text{pentru } s>d \\ m+1 & \text{pentru } a_m=x \\ \text{gasit}(s,m+1,a,x) & \text{pentru } a_m > x \\ \text{gasit}(m-1,d,a,x) & \text{pentru } x > a_m \end{cases}$$

```

void citeste (int a[], int &n)
{int i; cout<<"n= "; cin>>n;
for (i=0;i<n;i++) {cout<<"a("<<i+1<<"= ";cin>>a[i];}}
void main()
{int x,n,a[20],poz; cout<<"x= "; cin>>x;
citeste(a,n); poz=gasit(0,n-1,a,x);
if (poz== -1) cout<<"Nu exista elementul";
    else cout<<"s-a gasit in pozitia "<<poz; }

```

Exemplul 6

Să se sorteze crescător un vector cu n numere reale. Pentru sortare s-a folosit metoda selecției directe. Funcția recursivă este **sort(a,i,j,n)**, unde **a** este vectorul, **i** și **j** indicii folosiți de metoda de sortare și **n** lungimea logică a vectorului. Cu indicele **i** se parcurge vectorul de la primul element până la penultimul (indicele **i** ia valori de la 0 până la $n-2$), iar cu indicele **j** se parcurge vectorul de la elementul următor elementului cu indicele **i** până la ultimul (indicele ia valori de la $i+1$ până la $n-1$). Se evaluează funcția **sort(a,0,1,n)**.

```

#include <iostream.h>
void citeste(float a[],int n)
{for (int i=0; i<n;i++) {cout<<"a("<<i+1<<"= ";cin>>a[i];}}
void scrie(float a[],int n)
{for (int i=0; i<n;i++) cout<<a[i]<<" ";}
void sort(float a[],int i,int j,int n)
{float aux;
if (i!=n-2 || j!=n-1)
{if (a[j]<a[i]) {aux=a[i]; a[i]=a[j]; a[j]=aux;}
if (j!=n-1) sort(a,i,j+1,n);
else sort(a,i+1,i+2,n);}}
void main()
{int n; float a[50]; cout<<"n= "; cin>>n;
citeste (a,n); sort(a,0,1,n); scrie(a,n);}

```

Temă



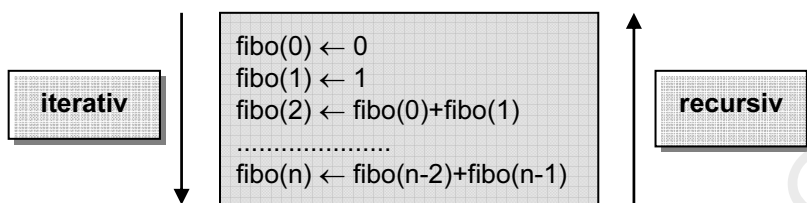
1. Pentru un vector cu elemente întregi, scrieți un subprogram recursiv, care calculează:
 - a. suma elementelor vectorului;
 - b. suma elementelor pozitive din vector;
 - c. media aritmetică a elementelor vectorului;
 - d. media aritmetică a elementelor pozitive din vector.
2. Scrieți un subprogram recursiv care să inverseze elementele unui vector în el însuși.
3. Scrieți un subprogram recursiv care să sorteze crescător elementele unui vector, folosind metoda bulelor.

1.2.6. Recursivitatea în cascadă

Se definește șirul lui Fibonacci, prin funcția **fibo:N→N**:

$$\text{fibo}(n) = \begin{cases} 0 & \text{pentru } n=0 \\ 1 & \text{pentru } n=1 \\ \text{fibo}(n-1)+\text{fibo}(n-2) & \text{pentru } n>1 \end{cases}$$

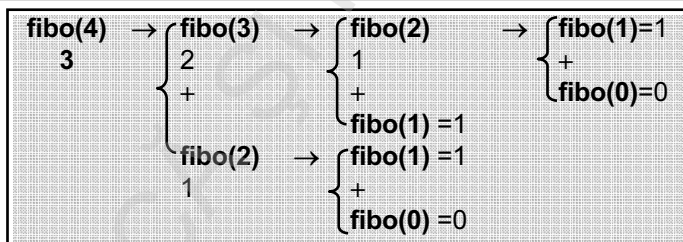
Algoritmul folosit poate fi descris astfel:



Programul care folosește algoritmul recursiv pentru calcularea termenului de ordinul n al șirului lui Fibonacci **execută autoapelul de două ori în cazul general** al soluției:

```
#include <iostream.h>
int fibo(int n)
{if (n<=1) return n;
  else return fibo(n-1)+fibo(n-2);}
void main()
{int n; cout<<"n= "; cin>>n;
 cout<<"Termenul "<<n<<" al sirului lui Fibonacci este ";
 cout <<fibo(n);}
```

Adâncimea recursivității pentru apelul $\text{fibo}(4)$ este 9. Dacă notăm cu $\text{ar}(n)$ adâncimea recursivității pentru apelul $\text{fibo}(n)$, observăm că:



$$\text{ar}(1) = \text{ar}(0) = 1$$

$$\text{ar}(2) = 1 + \text{ar}(1) + \text{ar}(0) = 1 + 1 + 1 = 3$$

$$\text{ar}(3) = 1 + \text{ar}(2) + \text{ar}(1) = 1 + 3 + 1 = 5$$

$$\text{ar}(4) = 1 + \text{ar}(3) + \text{ar}(2) = 1 + 5 + 3 = 9$$

$$\text{ar}(5) = 1 + \text{ar}(4) + \text{ar}(3) = 1 + 9 + 5 = 15$$

$$\text{ar}(6) = 1 + \text{ar}(5) + \text{ar}(4) = 1 + 15 + 9 = 25$$

$$\text{ar}(7) = 1 + \text{ar}(6) + \text{ar}(5) = 1 + 25 + 15 = 41$$

$$\dots\dots\dots$$

$$\text{ar}(n) = 1 + \text{ar}(n-1) + \text{ar}(n-2)$$



Scrieți un program prin care calculați adâncimea recursivității pentru apelul $\text{fibo}(n)$. Calculați adâncimea recursivității pentru următoarele apeluri: $\text{fibo}(10)$, $\text{fibo}(15)$, $\text{fibo}(20)$, $\text{fibo}(25)$, $\text{fibo}(30)$, $\text{fibo}(40)$, $\text{fibo}(50)$. Ce constatați?

Observație:

Acest tip de recursivitate se numește **recursivitate în cascadă** și este un exemplu în care programul iterativ este mult mai eficient decât programul recursiv. Ca să se poată executa $\text{fibo}(n)$ trebuie să se cunoască valoarea returnată de $\text{fibo}(n-1)$ și $\text{fibo}(n-2)$. Aceasta înseamnă că mai întâi se calculează valoarea lui $\text{fibo}(n-1)$, după care se reia calculul pentru valoarea lui $\text{fibo}(n-2)$, rezervându-se câte o zonă în stivă pentru fiecare apel în care se salvează contextul funcției. Implementarea recursivă este ineficientă pentru valori mari ale lui n . Pentru exemplificare, se pot executa ambele versiuni (cea recursivă și cea iterativă), pentru $n=100$.

Pentru a reduce adâncimea recursivității programului care calculează termenul n al șirului lui Fibonacci, se pornește de la observația că

$$\text{fibo}(n,a,b) = \begin{cases} a & \text{pentru } n=0 \\ \text{fibo}(n-1,b,a+b) & \text{pentru } n \geq 1 \end{cases}$$

termenul n este egal cu suma dintre termenul predecesor (a) și termenul antepredecessor (b), obținându-se definiția recursivă alăturată. Termenul n al șirului lui Fibonacci se va calcula prin evaluarea funcției $\text{fib}(n, 0, 1)$ – unde, pentru primul apel, a și b au valoarea 0 și respectiv 1, corespunzătoare primilor doi termeni ai șirului. De exemplu, dacă $n=4$, funcția se calculează astfel: $\text{fib}(4, 0, 1) = \text{fib}(3, 1, 1) = \text{fib}(2, 1, 2) = \text{fib}(1, 2, 3) = \text{fib}(0, 3, 5) = 3$

Temă



1. Care este adâncimea recursivității pentru apelul $\text{fib}(9)$? Scrieți un program în care implementați algoritmul recursiv definit mai sus. Executați cele două programe care implementează algoritmi recursivi, pentru $n=50$, și comparați timpii de execuție.
2. Scrieți un subprogram recursiv care să verifice dacă două numere a și b citite de la tastatură sunt termeni consecutivi ai șirului lui Fibonacci.

Un alt exemplu de recursivitate în cascadă este algoritmul pentru rezolvarea următoarei probleme. Fie a_n, b_n, c_n trei șiruri, cu $n \geq 0$, definite astfel:

$$a(n) = \begin{cases} 1 & \text{pentru } n=0 \\ 3 & \text{pentru } n=1 \\ 2 \times a(n-1) - a(n-2) & \text{pentru } n > 1 \end{cases}$$

$$b(n) = \begin{cases} 4 & \text{pentru } n=0 \\ 2 & \text{pentru } n=1 \\ 2 \times b(n-2) - b(n-1) & \text{pentru } n > 1 \end{cases}$$

$$\text{și } c(n) = a(n) + b(n).$$

Următorul program calculează termenul $c(n)$, n fiind introdus de la tastatură.

```
#include <iostream.h>
int a(int n)
{if(!n) return 1;
 else if (n==1) return 3;
  else return 2*a(n-1)-a(n-2);}
int b(int n)
{if (!n) return 4;
 else if (n==1) return 2;
  else return 2*b(n-2)-b(n-1);}
void main()
{int n; cout<<"n= "; cin>>n; cout<<"c ("<<n<<" ) = "<<a(n)+b(n);}
```

Temă



Care este adâncimea recursivității pentru apelul $c(4)$? Dar pentru apelul $c(7)$? Scrieți un program prin care calculați adâncimea recursivității pentru apelul $c(n)$. Scrieți o funcție recursivă mai eficientă, prin care să reduceți adâncimea recursivității.

1.2.7. Recursivitatea directă și indirectă

În funcție de procesele care se autoapelează, există două tipuri de recursivitate:

- **recursivitatea directă**
- **recursivitatea indirectă**

Recursivitatea directă. În acest tip de recursivitate, există un singur proces, A , care își generează singur existența, pe baza unor condiții. Astfel, dacă în corpul unui subprogram A se întâlnește un apel al aceluiași subprogram A , înseamnă că subprogramul este direct recursiv.

```
void A ()
{
.....
  A ();
.....
}
```

Recursivitatea indirectă. În acest tip de recursivitate, există două procese, A și B, care își generează reciproc apariția. Astfel, dacă în corpul unui subprogram A se întâlnește un apel al subprogramului B, care la rândul său apelează subprogramul A, înseamnă că subprogramele sunt indirect recursive. În acest caz, trebuie să fie folosită obligatoriu declararea prin **prototip** a subprogramelelor.

Exemplul 1

Să se afișeze termenul n din următoarele șiruri:

$$\begin{aligned} a_n &= a_{n-1} + b_{n-1} \\ b_n &= (b_{n-1})^2 - (a_{n-1})^2 \end{aligned}$$

Numărul n și primii termeni, a_0 și b_0 , se citesc de la tastatură. Pentru $n=2$, $a(0)=1$ și $b(0)=2$, calculul funcțiilor $a(2)$ și $b(2)$ se desfășoară astfel:

$$\begin{array}{l} a(2) \rightarrow \left\{ \begin{array}{l} a(1) \rightarrow \left\{ \begin{array}{l} a(0) = 1 \\ + \\ b(0) = 2 \end{array} \right. \\ + \\ b(1) \rightarrow \left\{ \begin{array}{l} [b(0)]^2 = 2^2 = 4 \\ - \\ [a(0)]^2 = 1^2 = 1 \end{array} \right. \end{array} \right. \\ 6 \end{array}$$

$$\begin{array}{l} b(2) \rightarrow \left\{ \begin{array}{l} [b(1)]^2 \rightarrow \left\{ \begin{array}{l} [b(0)]^2 = 2^2 = 4 \\ - \\ [a(0)]^2 = 1^2 = 1 \end{array} \right. \\ - \\ [a(1)]^2 \rightarrow \left\{ \begin{array}{l} a(0) = 1 \\ + \\ b(0) = 2 \end{array} \right. \end{array} \right. \\ 0 \end{array}$$

```
#include <iostream.h>
#include <math.h>
double an(int, float, float); //prototipul funcției an()
double bn(int, float, float); //prototipul funcției bn()
void main()
{int n; double a,b;
 cout<<"n= ";cin>>n; cout<<"a= ";cin>>a; cout<<"b= ";cin>>b;
 cout<<"a("<<n<<" )= "<<an(n,a,b)<<endl;
 cout<<"b("<<n<<" )= "<<bn(n,a,b);}
double an(int n, float a, float b)
{if(!n) return a+b;
 else return an(n-1,a,b)+bn(n-1,a,b);}
double bn(int n, float a, float b)
{if (!n) return b*b-a*a;
 else return pow(bn(n-1,a,b),2)-pow(an(n-1,a,b),2);}
```

Exemplul 2

Se știe că un cioban are o oaie. După un an oaia face o mieluță. După un an, mieluța devine mioară. După un an, mioara devine oaie. Câte oi, mioare și mieluțe are ciobanul după n ani, presupunând că nu moare nici un animal și că fiecare oaie face în fiecare an câte o mieluță?

În tabelul următor se poate observa evoluția animalelor pe parcursul anilor, considerându-se ca an de referință anul 0:

anul	0	1	2	3	4	5	6	7	8	9	10
oi	1	1	1	2	3	4	6	9	13	19	28
mieluțe	0	1	1	1	2	3	4	6	9	13	19
mioare	0	0	1	1	1	2	3	4	6	9	13

Se observă că, într-un an, numărul de oi este egal cu numărul de oi, plus numărul de mioare din anul precedent, numărul de mioare este egal cu numărul de mieluțe din anul precedent, iar numărul de mieluțe este egal cu numărul de oi din anul precedent. Se vor folosi următoarele funcții: x – pentru numărul de mieluțe, y – pentru numărul de mioare, și z – pentru numărul de oi. Aceste funcții sunt indirect recursive. Ele sunt definite alăturat.

$$z(n) = \begin{cases} 1 & \text{pentru } n=0 \\ z(n-1)+y(n-1) & \text{pentru } n>0 \end{cases}$$

```
#include <iostream.h>
```

```
int x(int n);
```

```
int y(int n);
```

```
int z(int n);
```

```
void main()
```

```
{int n; cout<<"ani= "; cin>>n;
```

```
cout<<"Mielute= "<<x(n)<<endl; cout<<"Mioare= "<<y(n)<<endl;
```

```
cout<<"oi= "<<z(n); }
```

```
int x(int n) {if (n==0) return 0; else return z(n-1); }
```

```
int y(int n) {if (n==0) return 0; else return x(n-1); }
```

```
int z(int n) {if (n==0) return 1; else return z(n-1)+y(n-1); }
```

$$x(n) = \begin{cases} 0 & \text{pentru } n=0 \\ z(n-1) & \text{pentru } n>0 \end{cases}$$

$$y(n) = \begin{cases} 0 & \text{pentru } n=0 \\ x(n-1) & \text{pentru } n>0 \end{cases}$$



Afișați termenul n din următoarele șiruri de medii aritmetico - geometrice (șirurile lui Gauss) – numărul n și primii termeni a_0 și b_0 se citesc de la tastatură ($a_0, b_0 > 0$):

$$a_n = (a_{n-1} + b_{n-1})/2 \text{ și}$$

$$b_n = \text{SQRT}(a_{n-1} \times b_{n-1})$$

Pentru rezolvarea acestei probleme, scrieți două programe: într-un program, implementați algoritmul iterativ – și în celălalt program, algoritmul recursiv. Arătați cum se execută programul recursiv pentru $n=2$, $a_0=4$ și $b_0=4$.

1.2.8. Avantajele și dezavantajele recursivității

Din exemplele prezentate se observă că, pentru orice algoritm recursiv folosit pentru rezolvarea unei probleme, există un algoritm iterativ echivalent, și invers. Problema care se pune este pe care dintre cei doi algoritmi îi veți alege pentru a rezolva o problemă.

Exemplul 1

Să se calculeze valorile funcției Manna Pnuell, definită alăturat. Definiția funcției este recursivă, și implementarea ei printr-un subprogram recursiv este foarte simplă.

$$f(x) = \begin{cases} x-1 & \text{pentru } x \geq 12 \\ f(f(x+2)) & \text{pentru } x < 12 \end{cases}$$

```
#include <iostream.h>
```

```
int mp(int n)
```

```
{if (n>=12) return n-1; else return mp(mp(n+2)); }
```

```
void main()
```

```
{int n; cout<<"n= "; cin>>n; cout<<mp(n); }
```

De exemplu, dacă $n=8$, funcția se calculează astfel:

$f(8) = f(f(8+2)) = f(f(10)) = f(f(f(10+2))) = f(f(f(12))) = f(f(12-1)) = f(f(11)) = f(f(f(11+2))) = f(f(f(13))) = f(f(13-1)) = f(f(12)) = f(12-1) = f(11) = f(f(11+2)) = f(f(13)) = f(13-1) = f(12) = 12-1 = 11$

iar, dacă $n=20$, funcția se calculează astfel: $f(20) = 20-1 = 19$

Pentru a rezolva această problemă cu ajutorul unui algoritm iterativ, va trebui să folosim o structură de date de tip stivă, care va fi prezentată ulterior.



Scrieți un program care implementează algoritmul recursiv pentru calcularea valorii **funcției Ackermann** definită alăturat (**ack**: $N \times N \rightarrow N$)
De exemplu, dacă $m=2$ și $n=1$, funcția se calculează astfel:

$ack(2,1) = ack(1,ack(2,0)) =$

$ack(1,ack(1,1)) = ack(1, ack(0,$

$ack(1,0))) =$

$= ack(1, ack(0, ack(0,1))) = ack(1,$

$ack(0,2)) = ack(1,3) = ack(0, ack(1,2)) =$

$= ack(0, ack(0, ack(1,1))) = ack(0, ack(0, ack(0, ack(1,0)))) = ack(0, ack(0, ack(0, ack(0,1)))) =$

$ack(0, ack(0, ack(1,12))) = ack(0, ack(0,3)) = ack(0, 4) = 5$

$$ack(m,n) = \begin{cases} n+1 & \text{pentru } m=0 \\ ack(m-1,1) & \text{pentru } n=0 \\ ack(m-1, ack(m, n-1)) & \text{pentru } m,n \neq 0 \end{cases}$$

Exemplul 2

Fiind dat $n \in N, n \geq 1$, să se genereze toate permutările mulțimii $\{1, 2, 3, \dots, n\}$.

Elementele unei permutări se generează în vectorul p . Pentru generarea tuturor permutărilor vom folosi următorul algoritm:

Pas 1. Se generează permutarea de 1 element: $\{1\}$.

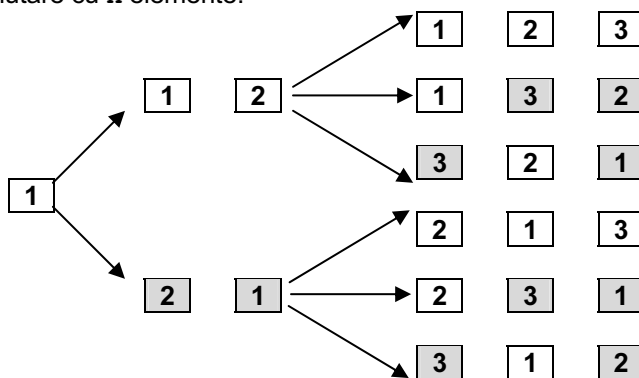
Pas 2. Pornind de la permutarea de 1 element, se generează cele 2 permutări de 2 elemente, astfel: atribuim celui de al doilea element valoarea 2, obținând permutarea $\{1, 2\}$, și apoi interschimbăm al doilea element cu primul element, obținând permutarea $\{2, 1\}$.

Pas n. Având generată o permutare cu $n-1$ elemente $\{a_1, a_2, \dots, a_{n-1}\}$ a mulțimii $\{1, 2, 3, \dots, n-1\}$, obținem n permutări ale mulțimii $\{1, 2, 3, \dots, n-1, n\}$ astfel:

Pas n.1. Atribuim elementului a_n valoarea n .

Pas n.2. Se interschimbă pe rând elementul a_n cu toate elementele a_1 , obținându-se câte o permutare cu n elemente.

Generarea permutărilor este un proces recursiv în care o permutare cu n elemente se generează pornind de la o permutare cu $n-1$ elemente, prin adăugarea unui nou element și interschimbarea valorii adăugate cu toate elementele permutării cu $n-1$ elemente.



```
#include <iostream.h>
```

```
unsigned n,p[10];
```

```
void afiseaza ()
```

```
{for (int i=1; i<=n;i++) cout<<p[i]<<" "; cout<<endl;}
```

```
void permut(unsigned i)
```

```
{unsigned j,aux;
```



```

if (i==n+1) afiseaza();
    else {p[i]=i; for (j=1;j<=i;j++)
        {aux=p[i]; p[i]=p[j]; p[j]=aux; permut(i+1);
        aux=p[i]; p[i]=p[j]; p[j]=aux;}}}

void main() {cout<<"n= "; cin>>n; permut(1);}

```

Temă



Arătați cum se execută programul pentru $n=4$. Scrieți un program prin care să implementați o soluție iterativă pentru această problemă.

Exemplul 2

Să se genereze toate partițiile numărului natural n . O partiție a unui număr natural este o mulțime de numere naturale a căror sumă este egală cu numărul n . Două partiții sunt diferite dacă diferă fie prin valori, fie prin ordinea lor. De exemplu, numărul 3 poate fi descompus astfel: $3 = 1+1+1 = 1+2 = 2+1$, și partițiile numărului 3 sunt $P_1 = \{1, 1, 1\}$, $P_2 = \{1, 2\}$ și $P_3 = \{2, 1\}$.

Partițiile numărului se generează în vectorul **part**, care se parcurge cu indicele **i**. Primul termen al partiției se generează în variabila **j**. Generarea partițiilor unui număr poate fi descrisă prin următorul algoritm:

Pas 1. Se inițializează primul termen al partiției $j=1$.

Pas 2. Se generează primul termen j al partiției P_i .

Pas 3. Se revine la **Pas2** pentru a genera o partiție a numărului rămas, $n-j$.

Pas 4. Dacă numărul rămas, $n-j$, devine 0, se termină generarea partiției P_i și se afișează partiția; altfel, se revine la **Pas3**.

Pas 5. Se incrementează cu 1 primul termen al partiției ($j=j+1$). Dacă $j>n$, se termină generarea tuturor partițiilor; altfel, se revine la **Pas2**.

Generarea partiției P_i a numărului n (**Pas3**) poate fi descrisă printr-un proces recursiv, care constă în generarea tuturor partițiilor numărului rămas, $n-j$.

```

#include <iostream.h>
unsigned n,part[50];

void scrie(unsigned m)
{cout<<"n= "<<part[0];
 for(unsigned i=1;i<m;i++) cout<<" + "<<part[i];
 cout<<endl;}

void partitie(unsigned i,unsigned n)
{for (unsigned j=1;j<=n;j++)
 {part[i]=j; if (j<n) partitie(i+1,n-j);
  else scrie(i+1);}}

void main()
{cout<<"n= "; cin>>n; partitie(0,n);}

```

Temă



Arătați cum se execută programul pentru $n=4$. Scrieți un program prin care să implementați o soluție iterativă pentru această problemă.

Observații

1. Din exemplele prezentate rezultă că soluțiile recursive sunt mult mai clare, mai scurte și mai ușor de urmărit. Soluția recursivă este mult mai avantajoasă decât cea iterativă, în următoarele cazuri:

- soluțiile problemei sunt definite recursiv;
- cerințele problemei sunt formulate recursiv.

În unele cazuri este foarte greu de definit o soluție iterativă, cum este cazul algoritmilor cu recursivitate indirectă, și atunci se preferă algoritmul recursiv.

2. Fiecare apel al unui subprogram recurent înseamnă încă o zonă de memorie rezervată pentru execuția subprogramului (variabilele locale și instrucțiunile). Pentru o adâncime mare a recursivității, algoritmi recursivi nu mai sunt eficienți, deoarece timpul de execuție crește, din cauza timpilor necesari pentru mecanismul de apel și pentru administrarea stivei de sistem.

Alegerea unuia dintre cei doi algoritmi – iterativ sau recursiv – pentru rezolvarea unei probleme, depinde de programator, în funcție de avantajele și dezavantajele fiecăruia.

Evaluate**Răspundeți:**

1. Care este valoarea returnată de funcția următoare, la apelul $f(10)$? Dar la apelul $f(100)$? Ce realizează această funcție?

```
int f(int n) {if(n==0) return 0; else return f(n-1)+n;}
```

2. Care este valoarea returnată de funcția următoare, la apelul $f(10)$? Dar la apelul $f(100)$? Ce realizează această funcție?

```
int f(int n) {if(n<=0) return n; else return f(n-2)+n;}
```

3. Care este valoarea returnată de funcția următoare, la apelul $f(5)$? Ce realizează această funcție?

```
int f(int n) {if(n<=0) return 1; else return 2*f(n-1);}
```

4. Care este valoarea returnată de funcția următoare, la apelul $f(5)$? Dar la apelul $f(25)$? Ce realizează această funcție?

```
int f(int n) {if(n<=1) return 0; else return f(n-1) + 2*n;}
```

5. Care este valoarea returnată de funcția următoare, la apelul $f(10)$? Dar la apelul $f(10)$? Ce realizează această funcție?

```
int f(int n)
{if(n==0) return 0;
 else if(n%2==0) return f(n-1)+n; else return f(n-1)-n;}
```

6. Care este valoarea returnată de funcția următoare, la apelul $f(3,5)$? Ce realizează această funcție?

```
int f(int a,int b)
{if (b==1) return a; else return a+f(a,b-1);}
```

7. Care este valoarea returnată de funcția următoare, la apelul $f(1536,2)$? Ce realizează această funcție?

```
int f(int n,int i)
{if (n<=i) return 1;
 else if(n%i==0) return 1+f(n/i,i); else return f(n,i+1);}
```

8. Ce se va afișa în urma execuției următorului program, dacă se citește de la tastatură 5?

```
#include <iostream.h>
void f(int n, int i)
{if(n>=1) {if (n%2==0) f(n-1,i); else f(n-1,i-1);
           cout<<i<<" ";}
void main() {int n,i; cout<<"n= "; cin>>n; f(2*n,n); }
```

9. Ce se va afișa în urma execuției următorului program, dacă se citește de la tastatură 10?

```
#include <iostream.h>
#include <iomanip.h>
void f(int n,int i, int j)
{if (i!=n+1){cout<<"* ";
            if (j==i) {cout<<endl; f(n,i+1,1);} else f(n,i,j+1);}}
void main() {int n; cout<<"n=";cin>>n; f(n,1,1);}
```

10. Ce se va afișa în urma execuției următorului program, dacă se citește de la tastatură 5?

```
#include <iostream.h>
#include <iomanip.h>
void tr(int i,int j,int n)
{if (i<=n) if(j<=i) {cout<<setw(3)<<j; tr(i,j+1,n);}
             else {cout<<endl; tr(i+1,1,n);}}
void main() {int n; cout<<"n= "; cin>>n; tr(1,1,n);}
```

11. Ce se va afișa în urma execuției următorului program, dacă se citește de la tastatură 3550? Dar dacă se citește 2125? Dar dacă se citește 1280? Ce realizează acest program?

```
#include <iostream.h>
int term(int n,int i)
{if (i<=n) return term(n,2*i); else return i/2;}
void scrie(int n)
{if (n>5) scrie(n-term(n,5));
 if (n<=5) {if (n!=0) cout<<n<<" ";}
            else cout<<term(n,5)<<" ";}
void main() {int n; cout<<"n= "; cin>>n; scrie(n);}
```

12. Fie a și n două numere naturale. Următoarele 5 variante de funcții matematice definesc recursiv algoritmul pentru calculul lui a^n . Implementați acești algoritmi recursivi cu ajutorul subprogramelor.

$$a^n = \begin{cases} 1 & \text{pentru } n=0 \\ a \times a^{n-1} & \text{pentru } n>0 \end{cases}$$

$$b) \quad a^n = \begin{cases} 1 & \text{pentru } n=0 \\ a & \text{pentru } n=1 \\ (a \times a)^{n/2} & \text{pentru } n \text{ par} \\ a \times (a \times a)^{(n-1)/2} & \text{pentru } n \text{ impar} \end{cases}$$

$$c) \quad a^n = \begin{cases} 1 & \text{pentru } n=0 \\ a & \text{pentru } n=1 \\ a^2 \times a^{n-2} & \text{pentru } n \text{ par} \\ a \times a^{n-1} & \text{pentru } n \text{ impar} \end{cases}$$

$$d) \quad a^n = \begin{cases} 1 & \text{pentru } n=0 \\ (a^{n/2})^2 & \text{pentru } n \text{ par} \\ a \times a^{n-1} & \text{pentru } n \text{ impar} \end{cases}$$

$$e) \quad a^n = \begin{cases} 1 & \text{pentru } n=0 \\ a & \text{pentru } n=1 \\ a^{n/2} \times a^{n/2} & \text{pentru } n \text{ par} \\ a \times (a^{(n-1)/2} \times a^{(n-1)/2}) & \text{pentru } n \text{ impar} \end{cases}$$

Dacă antetul subprogramului recursiv este:

```
unsigned long p(unsigned a, unsigned n)
```

calculați, pentru fiecare dintre cele cinci exemple, adâncimea recursivității, pentru apelul $p(2, 10)$. Precizați care variantă este mai eficientă.

13. Implementați în limbajul de programare funcțiile recursive, folosind modelul următor, care calculează și adâncimea recursivității. Executați programele și comparați valorile afișate de program cu cele calculate la problema 12.

```
#include <iostream.h>  
int f(int a, int n, int &q) //varianta a  
{q++; if(n==0) return 1; else return a*f1(a,n-1,q); }  
void main()  
{int a,n,q=0; cout<<"a="; cin>>a; cout<<"n=";cin>>n;  
cout<<a<<"**"<<n<<"="<<f(a,n,q)<<endl;  
cout<<"Adancimea recursivitatii= "<<q<<endl;}
```

Adevărat sau Fals:

1. Funcția următoare returnează, la apelul $f(5)$, valoarea 11:

```
int f(int n)  
{if (n<=1) return n; else return f(n-1)+2*f(n-2);}
```

2. Funcția următoare returnează, la apelul $\text{numar}(0)$, numărul de apariții ale unui element precizat, x , într-un vector a , de numere întregi, cu n elemente.

```
int numar(int i)  
{if (i==n-1) return a[n-1]==x; else return a[i]==x+numar(i+1);}
```

Alegeți:

1. Cu ce expresie trebuie completată secvența lipsă din funcția următoare, pentru ca $f(x)$ să returneze modulul numărului real x ?

```
float f(float x) {if (...) return x; else return f(-x);}
```

- a. $x > 0$ b. $x \geq 0$ c. $x < 0$ d. $x \leq 0$

2. Cu ce expresie trebuie completată secvența lipsă din funcția următoare, pentru ca $f(a,b)$ să returneze cel mai mare divizor comun a două numere naturale, a și b , folosind algoritmul lui Euclid?

```
float f(float a, float b)  
{if (b==0) return a; else return f(...);}
```

- a. $a, a/b$ b. $b, a/b$ c. $a, a\%b$ d. $b, a\%b$

3. Cu ce expresie trebuie completată secvența lipsă din funcția următoare, pentru ca $f(n, \text{sqrt}(n))$ să determine dacă un număr natural n este prim (dacă este prim, funcția returnează rezultatul 1)?

```
int f(int n, int i)  
{if (i==1) return 1; else return ...;}
```

- a. $n/i \neq 0 \ \&\& \text{prim}(n, i-1)$ b. $n\%i \neq 0 \ \&\& \text{prim}(n, i-1)$
c. $n/i \neq 0 \ \&\& \text{prim}(n, i+1)$ d. $n\%i \neq 0 \ \&\& \text{prim}(n, i+1)$

4. Cu ce expresie trebuie completată secvența lipsă din funcția următoare, pentru ca, prin apelul $f(n, q)$, un număr natural n , scris în baza 10, să se afișeze ca număr scris în baza q ?

```
void f(int n, int q)
{int cifra=n%q;
if (n>=q) f(...); cout<<cifra;}
```

- a. $q/n, q$ b. $q\%n, q$ c. $n/q, q$ d. $n\%q, q$

5. Rezultatul furnizat de apelul **f(9,51)** este:

```
int f(int a, int b);
{if (a>b) return 0; else return 1+f(a,b-a);}
```

- a. 1 b. 6 c. 5 d. 4

6. Funcția de la problema precedentă furnizează:

- a. restul împărțirii lui a la b b. câtul împărțirii lui a la b
c. restul împărțirii lui b la a d. câtul împărțirii lui b la a

7. Cu ce expresie trebuie completată secvența lipsă din funcția următoare, pentru ca **este(n-1)** să testeze dacă o valoare dată, **x**, există într-un vector **a**, de numere întregi, cu **n** elemente (dacă valoarea este găsită în vector, funcția returnează rezultatul **1**)?

```
int este(int i)
{if (...) return 0;
else if (a[i]==x) return 1; else return este(i-1);}
```

- a. $i==0$ b. $i==n-1$ c. $i<0$ d. $i>=n$

8. Cu ce expresie trebuie completată secvența lipsă din funcția următoare, pentru ca **este(0)** să testeze dacă într-un vector **a**, de numere întregi, cu **n** elemente, există cel puțin un număr pozitiv (dacă există, funcția returnează rezultatul **1**)?

```
int este(int i)
{if (...) return 0;
else if (a[i]>x) return 1; else return este(i+1);}
```

- a. $i==0$ b. $i==n-1$ c. $i<0$ d. $i>=n$

9. Programul următor va afișa pe ecran:

```
int a [20]={1,2,3,4,5,6,7,8,9,10};
int f(int i, int j);
{if (i>j) return 0;
else if (a==b) return a[i];
else return f(i, (i+j)/2) + f((i+j)/2+1, j);}
void main() {cout<<f(2,7); }
```

- a. 27 b. 35 c. 33 d. 37

10. Pentru funcția recursivă alăturată stabiliți care dintre următoarele expresii are valoarea 11:

- a. $1+f(4)$ b. $f(5)$ c. $f(6)$ d. $f(10)+1$

```
int f(int i)
{if (i<1) return 1;
else return 2+f(i-1);}
```

(Bacalaureat – Sesiunea iunie - iulie 2004)

11. Pentru a afișa configurația de asteriscuri de mai jos cu ajutorul subprogramului recursiv **proc**, se utilizează apelul:

```
***
**
*
```

- a. **proc(4,3)** b. **proc(4,4)** c. **proc(3,2)** d. **proc(3,3)**

```
void proc(int i, int j)
{cout<<"*";
if (i>1)
if (j>1) proc(i,j-1);
else {cout<<endl;
proc(i-1,i-1);}}
```

(Bacalaureat – Sesiunea iunie - iulie 2004)

12. Pentru definiția alăturată a subprogramului **sk**, stabiliți câte asteriscuri se afișează la apelul **sk(6)**:

a. 4 b. 2 c. un număr impar d. 6
(Bacalaureat – Simulare 2004)

```
void sk(unsigned int x)
{unsigned int x;
 for (i=1; i<=x/2; i++) sk(i);
 cout<<"*";}
```

13. Funcția recursivă **suma** trebuie definită astfel încât apelul **suma(n)** să returneze suma pătratelor perfecte mai mici sau egale cu **n**. Care este expresia cu care trebuie completată definiția funcției?

a. $j*j+suma(j*j-1)$
c. $j*j+suma(j)$

b. $j+suma(j*j-1)$
d. $j*j+suma(j-1)$

```
long suma(long i)
{if (i==0) return 0;
 else {long j=sqrt(i);
      return ...;}}
```

(Bacalaureat – Sesiunea iunie - iulie 2003)

14. Funcția **prim(int i)** returnează valoarea 1 dacă **i** este număr prim și valoarea 0 în caz contrar. Funcția recursivă **max** trebuie definită astfel încât apelul **max(n)** să returneze cel mai mare număr prim mai mic sau egal cu **n** sau valoarea 0 – în caz că nu există un astfel de număr. Care este expresia cu care trebuie completată definiția funcției?

a. $i-1$ b. $1+max(i-1)$

c. $prim(i-1)$ d. $max(i-1)$

```
int max(int i)
{ //1 nu este numar prim
 if (i<2) return 0;
 else if (prim(i)) return i;
 else return ...;}
```

(Bacalaureat – Sesiunea specială 2003)

Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra în **echipă**. Fiecare miniproiect va conține:

- implementarea iterativă și recursivă a subprogramelor;
- explicarea noțiunilor teoretice folosite pentru realizarea subprogramelor recursive;
- avantajele și dezavantajele fiecărei implementări.

1. Scrieți un subprogram care să calculeze combinații de **n** luate câte **k** – **C(n,k)** definite prin funcțiile recursive:

→ $C(n,k) = C(n-1,k) + C(n-1,k-1)$, cu $C(n,0) = C(n,n) = 1$ și $C(n,1) = n$

→ $C(n,k) = ((n-k+1)/k) * C(n,k-1)$, cu $C(n,0) = 1$

Calculați, pentru fiecare dintre subprogramele recursive, adâncimea recursivității, pentru **C(5,2)**. Care dintre implementări este mai eficientă?

- Se citește de la tastatură un număr natural **n**. Scrieți un subprogram care generează și afișează toate permutările circulare ale mulțimii $X = \{1, 2, 3, \dots, n\}$.
- Scrieți un subprogram care să calculeze valoarea unui polinom de gradul **n**, cu coeficienți întregi, într-un punct real **x**. Gradul polinomului, coeficienții lui și valoarea lui **x** se citesc de la tastatură.
- Se citesc trei numere întregi, **a**, **b** și **c**, care reprezintă coeficienții unei ecuații de gradul 2, și un număr natural **n**. Să se calculeze $S_n = x_1^n + x_2^n$, unde x_1 și x_2 sunt rădăcinile ecuației. Suma se calculează fără a se rezolva ecuația de gradul 2.
- Scrieți un program care să afișeze termenul **n** din următoarele șiruri, cu $a_0 = b_0 = 1$:

$$a_n = a_{n-1} + b_{n-1}$$

$$b_n = a_{n-1} - b_{n-1}$$

Calculați adâncimea recursivității pentru apelurile **a(10)**, **b(10)**, **a(11)**, **b(11)**, **a(n)** și **b(n)**.

6. Să se verifice dacă un număr n este superprim. Un număr este **superprim** dacă toate prefixele sale sunt numere superprime (de exemplu, numărul 171 este superprim, deoarece toate prefixele sale – 1, 17 și 171 – sunt numere prime). Vor fi implementate recursiv atât subprogramul care testează dacă un număr este prim, cât și subprogramul care extrage toate prefixele numărului.
7. Să se genereze toate numerele superprime dintr-un interval $[a, b]$ (valorile pentru a și b se citesc de la tastatură).
8. Să se genereze toate numerele binare cu n cifre care au m cifre de 0 și restul cifrelor 1. Valorile pentru n și m se citesc de la tastatură. Combinațiile posibile de m cifre de 0 și $n-m$ cifre de 1 se vor genera recursiv. Din mulțimea de combinații posibile se vor alege numai acelea care pot forma un număr cu n cifre.
9. Să se genereze toate submulțimile mulțimii $\{1, 2, 3, \dots, n\}$. Mulțimile generate sunt $A_k = \{a_{k1}, a_{k2}, a_{k3}, \dots, a_{ki}\}$, cu i care poate lua valori de la 1 la n , și $a_{k1} < a_{k2} < a_{k3} < \dots < a_{ki}$. De exemplu, pentru $n=3$ submulțimile sunt: $\{1\}$, $\{1, 2\}$, $\{1, 3\}$, $\{1, 2, 3\}$, $\{2\}$, $\{2, 3\}$ și $\{3\}$.

Indicație. Elementele submulțimii A_k se generează în vectorul a , care se parcurge cu indicele i . Elementul curent al unei submulțimi se generează în variabila j . Generarea submulțimii A_k începe cu alegerea primului element, a_{k1} , și generarea apoi a submulțimii $\{a_{k2}, a_{k3}, \dots, a_{ki}\}$. Pentru generarea acestei submulțimi, se procedează ca și pentru submulțimea A_k . Așadar, generarea submulțimilor poate fi descrisă printr-un proces recursiv: se alege pe rând o valoare j pentru a_{k1} începând cu prima valoare (1) până la ultima valoare (n). Pentru fiecare valoare aleasă pentru j , se obține o mulțime cu un element. Pornind de la fiecare dintre aceste submulțimi, se generează recursiv noi submulțimi, prin adăugarea a câte unui nou element, începând cu elementul cu valoarea $j+1$. Generarea recursivă a acestui grup de submulțimi se termină atunci când s-a adăugat elementul cu valoarea n .

10. Să se genereze partițiile mulțimii $X = \{1, 2, 3, \dots, n\}$, cu $n \in \mathbb{N}^*$. O partiție a mulțimii X se formează prin descompunerea mulțimii într-o reuniune de mulțimi disjuncte: $X = X_1 \cup X_2 \cup \dots \cup X_i$, unde $i \in \mathbb{N}^*$ și $i \leq n$. Mulțimile X_i au proprietățile:

→ $X_i \subset X$, pentru $\forall i, i \in \mathbb{N}^*, i \leq n$, și

→ $X_j \cap X_k = \emptyset$, pentru $\forall j$ și $k \in \mathbb{N}^*, j \leq i, k \leq i$ și $j \neq k$.

Indicație. Din definiția partiției unei mulțimi rezultă că un element k se poate găsi numai într-o submulțime X_i . Partițiile pot fi codificate într-un vector p , ale cărui elemente p_k reprezintă indicele i al submulțimii X_i din care face parte elementul k . Elementul 1 apare întotdeauna numai în X_1 , elementul 2 apare întotdeauna numai în X_1 și X_2 , ..., elementul k apare întotdeauna numai în $X_1, X_2, \dots$ și X_k . Generarea unei partiții înseamnă de fapt generarea unui set de valori pentru vectorul p . De exemplu, pentru $X = \{1, 2, 3\}$ partițiile sunt cele prezentate în tabelul alăturat.

partițiile mulțimii $X = \{1, 2, 3\}$		vectorul p		
		p_1	p_2	p_3
$\{1, 2, 3\}$	$X = X_1$	1	1	1
$\{1, 2\} \cup \{3\}$	$X = X_1 \cup X_2$	1	1	2
$\{1, 3\} \cup \{2\}$	$X = X_1 \cup X_2$	1	2	1
$\{1\} \cup \{2, 3\}$	$X = X_1 \cup X_2$	1	2	2
$\{1\} \cup \{2\} \cup \{3\}$	$X = X_1 \cup X_2 \cup X_3$	1	2	3

1.3. Analiza algoritmilor

Prin **analiza unui algoritm** se identifică resursele necesare pentru executarea algoritmului: **timpul de execuție și memoria**.

Analiza algoritmilor este necesară atunci când există mai mulți algoritmi pentru rezolvarea aceleiași probleme și trebuie ales algoritmul cel mai eficient.

Eficiența unui algoritm este evaluată prin timpul necesar pentru executarea algoritmului.

Pentru a compara din punct de vedere al eficienței doi algoritmi care rezolvă aceeași problemă, se folosește aceeași **dimensiune a datelor de intrare** – **n** (același număr de valori pentru datele de intrare).

Timpul de execuție al algoritmului se exprimă prin numărul de **operații de bază** executate în funcție de **dimensiunea datelor de intrare: $T(n)$** .

Pentru a compara doi algoritmi din punct de vedere al timpului de execuție, trebuie să se stabilească unitatea de măsură care se va folosi, adică **operația de bază** executată în cadrul algoritmilor, după care, se numără de câte ori se execută operația de bază în cazul fiecărui algoritm.

Operația de bază este o operație elementară sau o succesiune de operații elementare a căror execuție nu depinde de valorile datelor de intrare.

Există algoritmi la care **timpul de execuție** depinde de **distribuția datelor de intrare**. Să considerăm doi algoritmi de sortare a unui vector cu **n** elemente – algoritmul de sortare prin metoda selecției directe și algoritmul de sortare prin metoda bulelor – și ca operație de bază **comparația**. Dacă, în cazul primului algoritm, timpul de execuție nu depinde de distribuția datelor de intrare (modul în care sunt aranjate elementele vectorului înainte de sortarea lui), el fiind $T(n) = \frac{n \times (n-1)}{2}$, în cazul celui de al doilea algoritm, timpul de exe-

cuție depinde de distribuția datelor de intrare (numărul de execuții ale structurii repetitive **while** depinde de modul în care sunt aranjate elementele vectorului înainte de sortare). În cazul în care numărul de execuții ale operațiilor elementare depinde de distribuția datelor de intrare, pentru analiza algoritmului se folosesc:

- **timpul maxim de execuție** – timpul de execuție pentru cazul cel mai nefavorabil de distribuție a datelor de intrare; în cazul sortării prin metoda bulelor, cazul cel mai nefavorabil este atunci când elementele vectorului sunt aranjate în ordine inversă decât aceea cerută de criteriul de sortare;
- **timpul mediu de execuție** – media timpilor de execuție pentru fiecare caz de distribuție a datelor de intrare.

Deoarece, în analiza eficienței unui algoritm, se urmărește comportamentul lui pentru o dimensiune mare a datelor de intrare, pentru a compara doi algoritmi, din punct de vedere al eficienței, este suficient să se ia în considerare numai factorul care determină **timpul de execuție** – și care este denumit **ordinul de complexitate**.

Ordinul de complexitate al unui algoritm îl reprezintă timpul de execuție – estimat prin ordinul de mărime al numărului de execuții ale operației de bază: $O(f(n))$, unde **f(n)** reprezintă termenul determinant al timpului de execuție, **T(n)**.

De exemplu, dacă, pentru algoritmul de sortare prin metoda selecției directe, timpul de execuție este $T(n) = \frac{n \times (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$, ordinul de complexitate al algoritmului este $O(n^2)$, deoarece în calcularea lui se ia în considerare numai factorul determinant din timpul de execuție.

În funcție de **ordinul de complexitate**, există următoarele **tipuri de algoritmi**:

Ordin de complexitate	Tipul algoritmului
$O(n)$	Algoritm liniar.
$O(n^m)$	Algoritm polinomial. Dacă $m=2$, algoritmul este pătratic , iar dacă $m=3$, algoritmul este cubic .
$O(k^n)$	Algoritm exponențial. De exemplu, 2^n , 3^n etc. Algoritmul de tip $O(n!)$ este tot de tip exponențial, deoarece: $1 \times 2 \times 3 \times 4 \times \dots \times n > 2 \times 2 \times 2 \times \dots \times 2 = 2^{n-1}$.
$O(\log n)$	Algoritm logaritmic.
$O(n \log n)$	Algoritm liniar logaritmic.

De exemplu, algoritmul de sortare prin metoda selecției directe este un algoritm **pătratic**.

Ordinul de complexitate este determinat de structurile repetitive care se execută cu mulțimea de valori pentru datele de intrare. În cazul structurilor repetitive imbricate, ordinul de complexitate este dat de produsul dintre numărul de repetiții ale fiecărei structuri repetitive.

Structura repetitivă	Numărul de execuții ale corpului structurii	Tipul algoritmului
for (i=1; i<=n; i=i+k) { }	$f(n)=n/k \rightarrow O(n)=n$	Linier
for (i=1; i<=n; i=i*k) { }	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
for (i=n; i>=n; i=i/k) { }	$f(n)=\log_k n \rightarrow O(n)=\log n$	Logaritmic
for (i=n; i<=n; i=i+p) { } for (j=n; j<=n; j=j+q) { }	$f(n)=(n/p) \times (n/q) = n^2/(p \times q) \rightarrow$ $O(n)=n^2$	Polinomial pătratic
for (i=n; i<=n; i=i++) { } for (j=i; j<=n; j=j++) { }	$f(n)=1+2+3+ \dots +n = (n \times (n+1))/2 \rightarrow$ $O(n)=n^2$	Polinomial pătratic

Temă

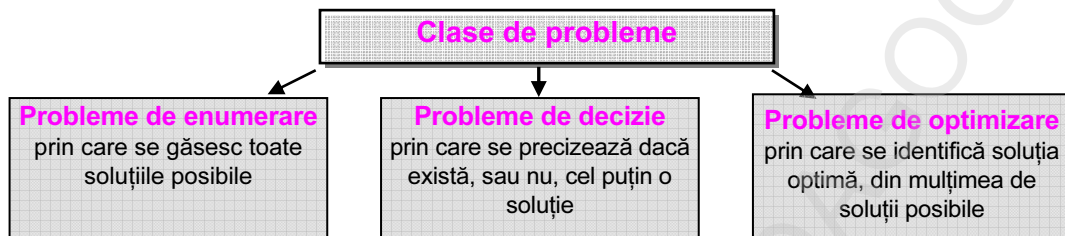


Determinați complexitatea următorilor algoritmi și precizați tipul algoritmului. Pentru fiecare algoritm se va considera dimensiunea datelor de intrare – **n**.

- determinarea valorii minime dintr-un șir de numere;
- inserarea unui element într-un vector, după un element cu valoare precizată;
- ștergerea dintr-un vector a unui element cu valoare precizată;
- stabilirea dacă un șir de numere conține numai numere distincte;
- sortarea unui vector folosind metoda bulelor;
- căutarea unui element cu valoare precizată, într-un vector nesortat;
- căutarea unui element cu valoare precizată, într-un vector sortat;
- determinarea tuturor permutărilor unei mulțimi de numere.

1.4. Metode de construire a algoritmilor

În funcție de procesul de calcul necesar pentru rezolvarea unei probleme, există următoarele clase de probleme:



Generarea tuturor permutărilor unei mulțimi de numere este o problemă de enumerare, căutarea unei valori precizate – într-un șir de numere – este o problemă de decizie, iar găsirea modalității de plată a unei sume s cu un număr minim de bancnote de valori date este o problemă de optimizare.

Pentru rezolvarea unei aceleiași probleme se pot folosi mai multe metode de construire a algoritmilor. Ați învățat deja că, pentru rezolvarea unei aceleiași probleme, puteți folosi un:

- **algoritm iterativ**;
- **algoritm recursiv**.

Soluțiile recursive sunt mult mai clare, mai scurte și mai ușor de urmărit. Alegerea algoritmului recursiv în locul celui iterativ este mai avantajoasă în cazul în care soluțiile problemei sunt definite recursiv sau dacă cerințele problemei sunt formulate recursiv.

Timpul de execuție al unui algoritm recursiv este dat de o formulă recursivă. De exemplu, pentru algoritmul de calculare a sumei primelor n numere naturale, funcția pentru timpul de execuție este prezentată alături, unde

$$T(n) = \begin{cases} \Theta(1) & \text{pentru } n=0 \\ \Theta(1)+T(n-1) & \text{pentru } n \neq 0 \end{cases}$$

$\Theta(1)$ reprezintă timpul de execuție al unei operații elementare de atribuire a unei valori sumei. Rezultă că $T(n)=(n+1) \times \Theta(1)$, iar ordinul de complexitate al algoritmului este $O(n)$, la fel ca și cel al algoritmului iterativ. În cazul implementării recursive, fiecare apel al unui subprogram recurent înseamnă încă o zonă de memorie rezervată pentru execuția subprogramului (variabilele locale și instrucțiunile). Din această cauză, în alegerea între un algoritm iterativ și un algoritm recursiv trebuie ținut cont nu numai de ordinul de complexitate, dar și de faptul că, pentru o **adâncime mare a recursivității**, algoritmi recursivi nu mai sunt eficienți, deoarece timpul de execuție crește, din cauza timpilor necesari pentru mecanismul de apel și pentru administrarea stivei de sistem.

Veți învăța noi **metode de construire** a algoritmilor – care vă oferă avantajul că prezintă fiecare o **metodă generală de rezolvare care se poate aplica unei clase de probleme**:

- **metoda backtracking**;
- **metoda divide et impera**.

Fiecare dintre aceste metode de construire a algoritmilor se poate folosi pentru anumite clase de probleme, iar în cazul în care – pentru o aceeași clasă de probleme – se pot folosi mai multe metode de construire a algoritmilor, criteriul de alegere va fi **eficiența** algoritmului.

1.5. Metoda backtracking

1.5.1. Descrierea metodei backtracking

Metoda backtracking se poate folosi pentru problemele în care trebuie **să se genereze toate soluțiile**, o **soluție** a problemei putând fi dată de un **vector**:

$$\mathbf{S} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$$

ale cărui elemente aparțin, fiecare, unor mulțimi finite $\mathbf{A}_i$ ($\mathbf{x}_i \in \mathbf{A}_i$), iar asupra elementelor unei soluții există anumite **restricții**, specifice problemei care trebuie rezolvată, numite **condiții interne**. Mulțimile $\mathbf{A}_i$ sunt mulțimi ale căror elemente sunt în relații bine stabilite. Mulțimile $\mathbf{A}_i$ pot să coincidă – sau nu. Pentru a găsi toate soluțiile unei astfel de probleme folosind o **metodă clasică de rezolvare**, se execută următorul algoritm:

PAS1. Se generează toate elementele produsului cartezian $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \dots \times \mathbf{A}_n$.

PAS2. Se verifică fiecare element al produsului cartezian, dacă îndeplinește **condițiile interne** impuse ca să fie soluție a problemei.

Studiu de caz

Scop: identificarea problemelor pentru care **trebuie enumerate toate soluțiile**, fiecare soluție fiind formată din n elemente $\mathbf{x}_i$, care aparțin fiecare unor mulțimi finite $\mathbf{A}_i$ și care trebuie să respecte anumite **condiții interne**.

Enunțul problemei 1: Să se genereze toate permutările mulțimii $\{1, 2, 3\}$.

Cerința este de a enumera toate posibilitățile de generare a **3** numere naturale din mulțimea $\{1,2,3\}$, astfel încât numerele generate să fie distincte (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu **3** elemente, $\mathbf{S} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$, în care elementul $\mathbf{x}_i$ reprezintă numărul care se va găsi, în permutare, pe poziția i , iar mulțimea $\mathbf{A}_i$ reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i . În acest exemplu mulțimile $\mathbf{A}_i$ coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$\mathbf{A}_i = \{1, 2, 3\} = \mathbf{A}$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 = \mathbf{A} \times \mathbf{A} \times \mathbf{A} = \mathbf{A}^3$, adică mulțimea:

$$\{(1,1,1), (1,1,2), (1,1,3), (1,2,1), \dots, (3,3,2), (3,3,3)\}$$

după care, se va verifica fiecare element al mulțimii dacă este o soluție a problemei, adică dacă cele trei numere dintr-o soluție sunt distincte. Soluțiile obținute sunt:

$$\{(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)\}$$

Enunțul problemei 2: Să se genereze toate aranjamentele de 2 elemente ale mulțimii $\{1, 2, 3\}$.

Cerința este de a enumera toate posibilitățile de generare a **2** numere naturale, din mulțimea $\{1,2,3\}$, astfel încât numerele generate să fie distincte (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu **2** elemente, $\mathbf{S} = \{\mathbf{x}_1, \mathbf{x}_2\}$, în care elementul $\mathbf{x}_i$ reprezintă numărul care se va găsi, în aranjament, pe poziția i , iar mulțimea $\mathbf{A}_i$ reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i . Și în acest exemplu, mulțimile $\mathbf{A}_i$ coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$\mathbf{A}_i = \{1, 2, 3\} = \mathbf{A}$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $\mathbf{A}_1 \times \mathbf{A}_2 = \mathbf{A} \times \mathbf{A} = \mathbf{A}^2$, adică mulțimea:

$$\{(1,1), (1,2), (1,3), (2,1), \dots, (3,2), (3,3)\}$$

după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele două numere dintr-o soluție sunt distincte. Soluțiile obținute sunt:

$$\{(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)\}$$

Enunțul problemei 3: Să se genereze toate combinațiile de 2 elemente ale mulțimii $\{1, 2, 3\}$.

Cerința este de a enumera toate posibilitățile de generare a 2 numere naturale din mulțimea $\{1,2,3\}$, astfel încât numerele generate să fie distincte (**condiția internă** a soluției), iar soluțiile obținute să fie distincte. Două soluții sunt considerate distincte dacă nu conțin aceleași numere. O soluție a acestei probleme va fi un vector cu 2 elemente, $S = \{x_1, x_2\}$, în care elementul x_i reprezintă numărul care se va găsi în combinație pe poziția i , iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i . Și în acest exemplu, mulțimile A_i coincid. Ele au aceleași 3 elemente, fiecare element reprezentând un număr.

$$A_i = \{1, 2, 3\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 = A \times A = A^2$, adică mulțimea:

$$\{(1,1), (1,2), (1,3), (2,1), \dots, (3,2), (3,3)\}$$

după care, se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele două numere dintr-o soluție sunt distincte și dacă soluția obținută este distinctă de soluțiile obținute anterior. Soluțiile obținute sunt: $\{(1,2), (1,3), (2,3)\}$

Enunțul problemei 4: Să se genereze toate permutările mulțimii $\{1,2,3,4\}$ care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

Cerința este de a enumera toate posibilitățile de generare a 4 numere naturale, din mulțimea $\{1,2,3,4\}$, astfel încât numerele generate să fie distincte, iar 1 să nu se învecineze cu 3, și 2 să nu se învecineze cu 4 (**condiția internă** a soluției). O soluție a acestei probleme va fi un vector cu 4 elemente, $S = \{x_1, x_2, x_3, x_4\}$, în care elementul x_i reprezintă numărul care se va găsi în permutare pe poziția i , iar mulțimea A_i reprezintă mulțimea numerelor din care se va alege un număr pentru poziția i . În acest exemplu, mulțimile A_i coincid. Ele au aceleași 4 elemente, fiecare element reprezentând un număr: $A_i = \{1, 2, 3, 4\} = A$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times A_4 = A \times A \times A \times A = A^4$, adică mulțimea:

$$\{(1,1,1,1), (1,1,1,2), (1,1,1,3), (1,1,1,4), \dots, (4,4,4,3), (4,4,4,4)\}$$

după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele patru numere dintr-o soluție sunt distincte, și dacă 1 nu se învecinează cu 3, iar 2 cu 4. Soluțiile obținute sunt:

$$\{(1,2,3,4), (1,4,3,2), (2,1,4,3), (2,3,4,1), (3,2,1,4), (3,4,1,2), (4,1,2,3), (4,3,2,1)\}$$

Enunțul problemei 5: Să se aranjeze, pe tabla de șah, opt dame care nu se atacă între ele (problema celor 8 dame).

Cerința este de a enumera toate posibilitățile de aranjare a 8 dame, pe o tablă de șah cu dimensiunea 8×8 (8 linii și 8 coloane), astfel încât toate cele 8 dame să nu se atace între ele (**condiția internă** a soluției). Deoarece nu se pot aranja două dame pe aceeași coloană (s-ar ataca între ele), înseamnă că pe fiecare coloană a tablei de șah se va pune o damă. O soluție a acestei probleme va fi un vector cu 8 elemente, $S = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$, în care elementul x_i reprezintă numărul liniei pe care se va pune dama în coloana i , iar mulțimea A_i reprezintă mulțimea liniilor pe care se poate aranja dama din coloana i . Și în acest caz mulțimile A_i coincid. Ele au aceleași opt elemente, fiecare element reprezentând un număr de linie:

$$A_i = \{1, 2, 3, 4, 5, 6, 7, 8\} = A$$

Dacă s-ar rezolva clasic această problemă, ar însemna să se genereze toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times \dots \times A_8 = A \times A \times A \times \dots \times A = A^8$, adică mulțimea: $\{(1,1,1,1,1,1,1,1), (1,1,1,1,1,1,1,2), (1,1,1,1,1,1,1,3), \dots, (8,8,8,8,8,8,8,7), (8,8,8,8,8,8,8,8)\}$ după care se va verifica fiecare element al mulțimii, dacă este o soluție a problemei, adică dacă cele opt numere dintr-o soluție pot reprezenta coloanele pe care pot fi aranjate damele, pe fiecare linie, astfel încât să nu se atace între ele. Soluțiile obținute sunt:

$\{(1,5,8,6,3,7,2,4), (1,6,8,3,7,4,2,5), (1,7,4,6,8,2,5,3), \dots, (8,3,1,6,2,5,7,4), (8,4,1,3,6,2,7,5)\}$

Observație. Metoda clasică de rezolvare a acestui tip de probleme necesită foarte multe operații din partea calculatorului, pentru a verifica fiecare element al produsului cartezian. Presupunând (pentru simplificare) că fiecare mulțime A_i are m elemente, atunci algoritmul de generare a elementelor produsului cartezian are complexitatea $O(\text{card}(A_1) \times \text{card}(A_2) \times \dots \times \text{card}(A_n)) = O(m \times m \times m \times \dots \times m) = O(m^n)$. Considerând că algoritmul prin care se verifică dacă un element al produsului cartezian este o soluție a problemei (respectă **condiția internă** a soluției) are complexitatea $O(p)$, atunci **complexitatea algoritmului** de rezolvare a problemei va fi $O(p \times m^n)$. De exemplu, în algoritmul de generare a permutărilor, complexitatea algoritmului de verificare a condiției interne este dată de complexitatea algoritmului prin care se verifică dacă numerele dintr-un șir sunt distincte. În acest algoritm, se parcurge șirul de m numere și, pentru fiecare număr din șir, se parcurge din nou șirul, pentru a verifica dacă acel număr mai există în șir. Complexitatea algoritmului este dată de cele două structuri **for** imbricate: $O(m^2) \rightarrow p = m^2$.



Metoda recomandată pentru acest gen de probleme este metoda **backtracking** sau **metoda căutării cu revenire**, prin care se reduce volumul operațiilor de găsire a tuturor soluțiilor.

Metoda **backtracking** construiește progresiv vectorul soluției, pornind de la primul element și adăugând la vector următoarele elemente – cu revenire la elementul anterior din vector – în caz de insucces. Elementul care trebuie adăugat se caută în mulțime, printre elementele care respectă condițiile interne.

Prin metoda **backtracking** se obțin **toate soluțiile problemei**, dacă ele există. Pentru exemplificarea modului în care sunt construite soluțiile, considerăm problema generării permutărilor mulțimii $\{1, 2, 3, \dots, n\}$ ($A_1 = A_2 = \dots = A_n = A = \{1, 2, 3, \dots, n\}$).

PAS1. Se alege primul element al soluției ca fiind primul element din mulțimea A . În exemplu, $x_1 = 1$, adică primul număr din permutare este 1.

PAS2. Se caută al doilea element al soluției (x_2). Pentru a-l găsi, se parcurg pe rând elementele mulțimii A și, pentru fiecare element i al mulțimii, se verifică dacă respectă **condițiile interne**. Căutarea continuă până când se găsește primul element din mulțimea A care îndeplinește **condiția internă**, după care se oprește. În exemplu, se caută numărul de pe a doua poziție a permutării, verificându-se dacă al doilea număr din permutare este diferit de primul număr (se parcurg primele două elemente ale mulțimii A). Se găsește elementul $x_2 = 2$, după care procesul de căutare se oprește.

PAS3. Se caută al treilea element al soluției (x_3). Căutarea va folosi același algoritm de la Pasul 2. În exemplu, se caută numărul din poziția a treia din permutare. Se găsește elementul $x_3 = 3$.

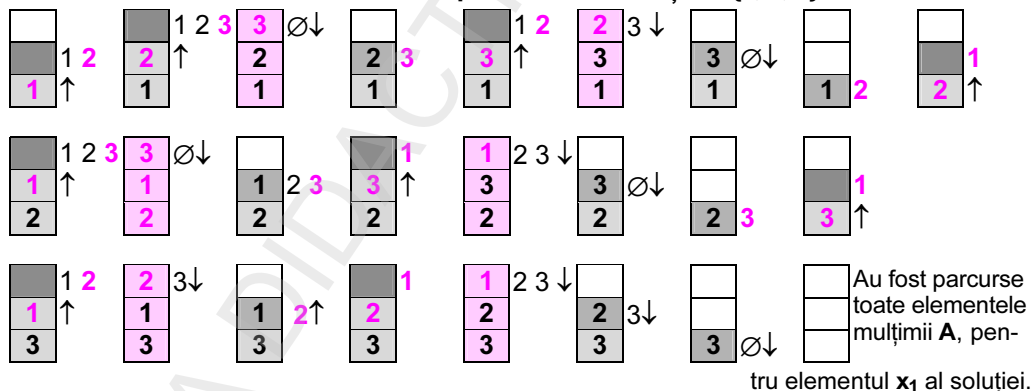
PAS4. Presupunând că s-au găsit primele k elemente ale soluției, $x_1, x_2, x_3, \dots, x_k$, se trece la căutarea celui de al $k+1$ -lea element al soluției, x_{k+1} . Căutarea se va face astfel: se atribuie, pe rând, lui x_{k+1} , elementele mulțimii A , până se găsește primul element i care îndeplinește condiția internă. În exemplu, condiția internă

este ca numărul din poziția $k+1$ a permutării să nu fie egal cu nici unul dintre numerele din pozițiile anterioare lui $k+1$. Pot să apară două situații:

- Există un element i , în mulțimea A , astfel încât $x_{k+1} = i$ să fie element al soluției problemei. În acest caz, se atribuie elementului x_{k+1} al soluției valoarea i , după care se verifică dacă s-a găsit soluția problemei. În exemplu, presupunem că pe nivelul $k+1$ s-a găsit numărul 4. Se verifică dacă s-au generat toate cele n elemente ale mulțimii S , adică dacă s-au găsit numere pentru toate cele n poziții din permutare ($k=n$). **Dacă** s-a găsit soluția problemei, **atunci** se afișează soluția; **altfel**, se caută următorul element al soluției, reluându-se operațiile de la Pasul 4.
- S-au parcurs toate elementele mulțimii A și nu s-a găsit nici un element i care să fie elementul x_{k+1} al soluției problemei. Înseamnă că trebuie să revenim la elementul k al soluției – x_k . Așadar, se consideră generate primele $k-1$ elemente ale soluției: $x_1, x_2, \dots, x_{k-1}$ și, pentru elementul x_k al soluției, se reia căutarea cu **următorul element din mulțimea A**, adică se reiau operațiile de la Pasul 4 pentru elementul x_k al soluției, însă nu cu primul element din mulțimea A , ci cu elementul din mulțimea A care se găsește imediat după cel care a fost atribuit anterior pentru elementul x_k al soluției. În exemplu, luând în considerare modul în care au fost generate primele k numere ale permutării, în poziția $k+1$ orice număr s-ar alege, el mai există pe una dintre cele k poziții anterioare – și se revine la elementul k , care presupunem că are valoarea 3. Se generează în această poziție următorul număr din mulțimea A (4) și se verifică dacă el nu mai există pe primele $k-1$ poziții ale permutării, iar dacă există, se generează următorul element din mulțimea A (5) ș.a.m.d.

PAS5. Algoritmul se încheie după ce au fost parcurse toate elementele mulțimii A pentru elementul x_1 al soluției. În exemplu, algoritmul se încheie după ce s-au atribuit, pe rând, valorile 1, 2, ..., n , elementului de pe prima poziție a permutării.

Generarea tuturor permutărilor mulțimii {1, 2, 3}



Observație. În metoda backtracking, dacă s-a găsit elementul x_k al soluției, elementului x_{k+1} al soluției i se atribuie o valoare numai dacă mai există o valoare care să îndeplinească **condiția de continuare** a construirii soluției – adică dacă, prin atribuirea acelei valori, se poate ajunge la o soluție finală pentru care condițiile interne sunt îndeplinite.

Temă



Desenați diagramele pentru generarea prin metoda **backtracking** a:

- tuturor aranjamentelor de 2 elemente ale mulțimii {1, 2, 3};
- tuturor combinațiilor de 2 elemente ale mulțimii {1, 2, 3};
- tuturor permutărilor mulțimii {1, 2, 3, 4} care îndeplinesc condiția că 1 nu este vecin cu 3, și 2 nu este vecin cu 4.

Algoritmul metodei **backtracking** poate fi generalizat pentru orice problemă care îndeplinește următoarele condiții:

1. Soluția problemei poate fi pusă sub forma unui vector $S = \{x_1, x_2, \dots, x_n\}$, ale cărui elemente x_i aparțin, fiecare, unei mulțimi A_i , astfel: $x_1 \in A_1, x_2 \in A_2, \dots, x_n \in A_n$.
2. Mulțimile A_i sunt **finite** iar elementele lor sunt **numere întregi** – și se găsesc într-o **ordine bine stabilită**.

Algoritmul backtracking este următorul:

PAS1. Se alege primul element al soluției S : $x_1 \in A_1$.

PAS2. **Cât timp** nu au fost parcurse toate elementele mulțimii A_1 (nu au fost găsite toate soluțiile) **execută**:

PAS3. Pentru fiecare element al soluției **execută**:

PAS4. Se presupune că s-au găsit primele k elemente ale soluției $(x_1, x_2, \dots, x_k)$ aparținând mulțimilor $A_1, A_2, A_3, \dots, A_k$, și se trece la căutarea celui de al $k+1$ -lea element al soluției, x_{k+1} , printre elementele mulțimii A_{k+1} . Căutarea se va face astfel: se atribuie, pe rând, lui x_{k+1} , elementele mulțimii A_{k+1} , până se găsește primul element care îndeplinește **condiția de continuare**.

PAS5. Dacă există un element a_i , în mulțimea A_{k+1} , astfel încât $x_{k+1} = a_i$ să aparțină soluției problemei, **atunci** se atribuie elementului x_{k+1} valoarea a_i și se trece la Pasul 7; **altfel**, se trece la Pasul 6.

PAS6. Deoarece s-au parcurs toate elementele mulțimii A_{k+1} și nu s-a găsit nici un element a_i care să îndeplinească condiția de continuare, se revine la elementul x_k și se consideră generate primele $k-1$ elemente ale soluției, $x_1, x_2, \dots, x_{k-1}$, și pentru elementul x_k se reia căutarea cu următorul element din mulțimea A_k , adică se reiau operațiile de la Pasul 4, pentru elementul x_k al soluției, însă nu cu primul element din mulțimea A_k ci cu elementul din mulțimea A_k care se găsește imediat după cel care a fost atribuit anterior elementului x_k .

PAS7. Se verifică dacă s-a găsit soluția problemei, adică dacă s-au găsit toate elementele mulțimii S . **Dacă** s-a găsit soluția problemei, **atunci** se afișează soluția; **altfel**, se trece la căutarea următorului element al soluției, reluându-se operațiile de la Pasul 4.

1.5.2. Implementarea metodei backtracking

Pentru implementarea metodei se folosesc următoarele structuri de date și subprograme.

Date și structuri de date

Pentru a memora elementele x_k ale soluției se folosește o structură de date de tip **vector**. Pentru indicele elementului care se adaugă la soluție se folosește variabila k . Când s-a găsit elementul x_k al soluției, se trece la pasul următor, prin incrementarea indicelui cu 1 ($k++$), pentru a căuta elementul x_{k+1} al soluției, iar pentru a reveni la elementul x_{k-1} al soluției se decrementează indicele cu 1 ($k--$). Inițial, vectorul va avea dimensiunea 1 ($k_{\min}=1$), corespunzătoare poziției de pornire, și va conține valoarea primului element al soluției. Pentru elementul x_k al soluției se va atribui o valoare din mulțimea A_k care poate fi un element al soluției: $s[k]=a[i]$. După parcurgerea completă a mulțimilor A_k , vectorul va avea dimensiunea n ($k_{\max}=n$), corespunzătoare numărului de elemente ale soluției. Indicele vectorului va fi inițial 1, la găsirea unei soluții va avea valoarea n , iar la terminarea algoritmului indicele va avea valoarea 0. Ne putem închipui vectorul soluție ca o construcție pe verticală, în care fiecare

element **k** al soluției reprezintă un nivel **k** care se adaugă la această construcție. Soluția se construiește de la nivelul **1** până la nivelul **n**.

Se mai folosesc următoarele variabile de memorie:

- **as** – pentru a ști dacă pentru elementul x_k al soluției mai există un succesor, adică dacă mai există un element în mulțimea A_k care ar putea fi elementul x_k al soluției (este o variabilă logică ce are valoarea 1 – *true*, dacă există succesor; altfel, are valoarea 0 – *false*);
- **ev** – pentru a ști dacă succesorul găsit respectă **condiția de continuare** și poate fi elementul x_k al soluției (este o variabilă logică ce are valoarea 1 – *true*, dacă succesorul este element al soluției; altfel, are valoarea 0 – *false*) și
- **n** – pentru dimensiunea soluției (numărul de elemente ale soluției, în cazul problemelor în care toate soluțiile au același număr de elemente).

```
int s[100]; //s=vectorul soluție
int n,k,ev,as; //k=indicele vectorului soluție
```

Pentru simplificarea implementării, toate aceste date și structuri de date sunt declarate globale deoarece valoarea indicelui elementului care se adaugă la soluție se va transmite mai ușor – între subprograme – ca variabilă globală.

Subprograme

Algoritmul va fi implementat prin:

- un subprogram, care va fi același pentru toți algoritmi de rezolvare prin metoda backtracking (**parte fixă**) – și care descrie **strategia generală backtracking** – și
- subprograme care au **aceeași semnificație** pentru toți algoritmi, dar al căror conținut diferă de la o problemă la alta, depinzând de **condițiile interne** ale soluției.

Semnificația **subprogramelor** folosite este (se va considera ca exemplu generarea permutărilor mulțimii $\{1, 2, 3, \dots, n\}$):

- Subprogramul **init** (funcție procedurală). Se inițializează elementul care urmează să se adauge la soluție (elementul de pe nivelul **k**). Acest element se inițializează cu o valoare care nu face parte din mulțimea A_k considerată, urmând ca în următorul pas al algoritmului să se atribue acestui element prima valoare din mulțimea A_k . În exemplu, elementul **k** al vectorului se va inițializa cu valoarea 0 (**s[k]=0**), urmând ca la pasul următor să i se atribue ca valoare 1, adică primul număr din mulțimea $\{1, 2, 3, \dots, n\}$.

```
void init()
{s[k]=0;}
```

- Subprogramul **succesor** (funcție operand). Verifică dacă mai există în mulțimea A_k un element pentru nivelul **k** al soluției (un succesor). Dacă mai există un succesor, se trece la următorul element din mulțimea A_k , iar funcția va returna valoarea 1 (*true*). Dacă nu mai există un succesor, funcția va returna valoarea 0 (*false*). Valoarea returnată de funcție se va atribui variabilei **as**. Inițial, valoarea variabilei de memorie **as** este 1 (*true*) – se presupune că mai există un succesor. În exemplu, subprogramul **succesor** va verifica dacă pentru poziția **k** din permutare mai există un număr. Dacă numărul **i** din poziția **k** este mai mic decât **n**, poziției **k** i se va atribui numărul următor, **i+1**, și funcția va returna valoarea 1 (*true*), iar dacă numărul din poziția **k** este **n**, înseamnă că pe această poziție din permutare nu mai poate fi pus nici un număr – și funcția va returna valoarea 0 (*false*).

```
int succesor()
{if (s[k]<n) {s[k]++; return 1;}
else return 0;}
```

→ Subprogramul **valid** (funcție operand). Verifică dacă valoarea atribuită elementului x_k al soluției îndeplinește **condiția de continuare**, adică poate fi considerată că face parte din soluția problemei (dacă succesorul găsit este element al soluției). Dacă este îndeplinită condiția (se evaluează expresia prin care este descrisă condiția), funcția va returna valoarea 1 (*true*); altfel, va returna valoarea 0 (*false*). Valoarea returnată de funcție se va atribui variabilei **ev**. Inițial, valoarea variabilei de memorie **ev** este 0 (*false*) – se presupune că succesorul găsit nu este elementul k al soluției. În exemplu, subprogramul **valid** va verifica dacă numărul din poziția k nu mai există în cele $k-1$ poziții anterioare. Dacă numărul nu îndeplinește această condiție, funcția va returna valoarea 0 (*false*).

```
int valid()
{for (int i=1; i<k; i++)
    if (s[i]==s[k]) return 0;
return 1;}
```

→ Subprogramul **solutie** (funcție operand). Verifică dacă s-au obținut toate elementele soluției. În exemplu, subprogramul **solutie** va verifica dacă au fost găsite toate cele n elemente ale soluției, adică dacă s-au găsit soluții de aranjare în permutare pentru toate cele n numere. Dacă s-a găsit soluția, subprogramul întoarce valoarea 1 (*true*); altfel, întoarce valoarea 0 (*false*).

```
int solutie()
{return k==n;}
```

→ Subprogramul **tipar** (funcție procedurală). Afișează elementele soluției. De obicei, afișarea soluției constă în afișarea valorilor din vectorul soluției.

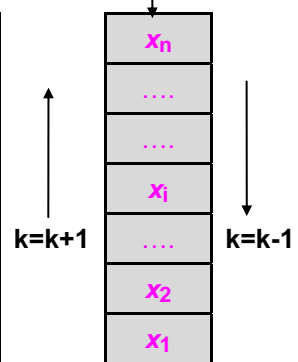
```
void tipar()
{for (int i=1; i<=n; i++) cout<<s[i]<<" ";
cout<<endl;}
```

Numărul de ordine al
elementelor soluției

n	a_{n1}	a_{n2}	...	a_{nj}	...	a_{nm}
....	Parcursarea elementelor se face cu					
....	subprogramul succesor					
i	a_{i1}	a_{i2}	...	a_{ij}	...	a_{im}
....						
2	a_{21}	a_{22}	...	a_{2j}	...	a_{2m}
1	a_{11}	a_{12}	...	a_{1j}	...	a_{1m}
	1	2	...	j		m

Numărul de ordine al elementelor din mulțimea A_k

Elementele soluției



Vectorul s : $s[k]$

Subprogramul **valid** verifică dacă este element al soluției

Subprogramul fix poate fi implementat **iterativ** sau **recursiv**.

Implementarea iterativă

```

void bt() //partea fixă a algoritmului
{
    k=1; //se inițializează indicele primului element al soluției
    init(); //se inițializează primul element al soluției
    while (k>0) //cât timp mai există valori în mulțimea A1
        //pentru primul nivel al soluției
    {
        as=1; ev=0;
        while(as && !ev) // cât timp are succesori și nu s-a găsit
            // elementul k al soluției
        {
            as=succesor(); // se caută succesori
            if(as) // dacă are succesori, atunci
                ev=valid(); // se verifică dacă este element al soluției
        }
        // se iese din structura repetitivă while dacă nu mai există
        // succesori sau dacă s-a găsit elementul soluției
        if(as) // dacă are succesori, atunci
        {
            if (solutie()) //dacă s-au obținut toate elementele soluției,
                tipar(); // atunci se afișează elementele soluției
            else {k++; // altfel, se trece la următorul nivel al soluției
                init();} // și se inițializează elementul
                // următor al soluției
        }
        else k--; // altfel se revine la nivelul anterior al soluției
    }
}

void main() { ... bt(); ... }

```

Implementarea recursivă – Prelucrările care se fac pentru nivelul k al soluției se fac și pentru nivelul $k+1$ al soluției – și aceste prelucrări pot fi apelate pentru elementul $k+1$ al soluției, iar trecerea de la elementul k al soluției la elementul $k+1$ al soluției se face prin apelul recursiv al acestor prelucrări. În algoritmul backtracking implementat iterativ revenirea la nivelul $k-1$ trebuie să se facă atunci când pe nivelul k al soluției nu se găsește o valoare care să îndeplinească **condițiile interne**. În cazul implementării recursive, **condiția de bază** este ca pentru nivelul k să nu se găsească o valoare care să îndeplinească **condițiile interne**. Când se ajunge la condiția de bază, încetează apelul recursiv și se revine la subprogramul apelant, adică la subprogramul în care se prelucra nivelul $k-1$ al soluției, iar în vector se vor regăsi valorile prelucrate anterior în acest subprogram. Deoarece apelul recursiv se face în funcție de valoarea indicelui elementului k al soluției, această valoare se va transmite – între subprograme – prin intermediul parametrilor de comunicație.

```

void bt(int k) //partea fixă a algoritmului
{
    init(k); //se inițializează elementul k al vectorului soluție
    while(succesor(k))
        //cât timp se găsește succesori pentru elementul k al soluției
    {
        if(valid(k)) dacă succesori sunt element al soluției
        {
            if(solutie(k)) //dacă s-au obținut toate elementele soluției
                tipar(k); // atunci se afișează elementele soluției
            else bt(k+1); //altfel se apelează subprogramul pentru a găsi
        }
        //elementul k+1 al soluției
    }
}

void main() { ... bt(1); ... }

```

Complexitatea algoritmului metodei backtracking

Dacă fiecare soluție are n elemente și fiecare mulțime A_i din care se alege un element al soluției are m elemente, atunci complexitatea algoritmului metodei backtracking este $O(\text{card}(A_1) \times \text{card}(A_2) \times \dots \times \text{card}(A_n)) = O(m \times m \times m \times \dots \times m) = O(m^n)$. Dacă numărul de elemente ale mulțimilor A_i este diferit – și notăm cu:

$$m_{\min} = \min(\text{card}(A_1), \text{card}(A_2), \dots, \text{card}(A_n))$$

$$m_{\max} = \max(\text{card}(A_1), \text{card}(A_2), \dots, \text{card}(A_n))$$

atunci complexitatea algoritmului va fi cuprinsă între o complexitate minimă $O(m_{\min}^n)$ și o complexitate maximă $O(m_{\max}^n)$. Rezultă că algoritmul metodei backtracking este un **algoritm exponențial**. Având o complexitate exponențială, metoda backtracking se recomandă numai dacă **nu se cunoaște un algoritm mai eficient**.

1.5.3. Probleme rezolvabile prin metoda backtracking

Metoda **backtracking** este recomandată în cazul problemelor care au următoarele caracteristici:

- se cere **găsirea tuturor soluțiilor posibile**;
- nu se cunoaște un algoritm mai eficient.

Alte exemple de probleme clasice care se pot rezolva folosind metoda **backtracking**:

- generarea tuturor elementelor unui produs cartezian;
- generarea tuturor partițiilor unui număr natural;
- generarea tuturor partițiilor unei mulțimi;
- generarea tuturor funcțiilor surjective;
- generarea tuturor funcțiilor injective;
- generarea tuturor posibilităților de plată a unei sume cu bancnote de valori date.

1.5.3.1. Generarea permutărilor

Prin asamblarea subprogramelor definite anterior, programul pentru generarea tuturor permutărilor mulțimii $\{1, 2, 3, \dots, n\}$ va fi:

Implementarea iterativă

```
#include<iostream.h>
int n,k,ev,as,s[100];
void init()
{s[k]=0;}
int sucesor()
{if (s[k]<n)
    {s[k]=s[k]+1; return 1;}
    else return 0;}
int valid()
{for(int i=1;i<k;i++)
    if (s[k]==s[i]) return 0;
return 1;}
int solutie()
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++)
    cout<<s[i]<<" ";
cout<<endl;}
```

Implementarea recursivă

```
#include<iostream.h>
int n, s[100];
void init(int k)
{s[k]=0;}
int sucesor(int k)
{if (s[k]<n)
    {s[k]=s[k]+1; return 1;}
    else return 0;}
int valid(int k)
{for(int i=1;i<k;i++)
    if (s[k]==s[i]) return 0;
return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++)
    cout<<s[i]<<" ";
cout<<endl;}
```

```

void bt()
{
    k=1;
    init();
    while (k>0)
    {
        as=1; ev=0;
        while(as && !ev)
        {
            as=succesor();
            if(as) ev=valid();
        }
        if(as)
        {
            if (solutie()) tipar();
            else {k++; init();}
        }
        else k--;}
}

void main()
{
    cout<<"n= "; cin>>n;
    bt();
}

```

```

void bt(int k)
{
    init(k);
    while(succesor(k))
    {
        if(valid(k))
        {
            if(solutie(k)) tipar();
            else bt(k+1);
        }
    }
}

void main()
{
    cout<<"n= "; cin>>n;
    bt(1);
}

```

Algoritmul de generare a permutărilor poate fi folosit și în alte probleme. De exemplu, să se genereze toate permutările mulțimii $\{1, 2, 3, \dots, n\}$ în care nu apar numere consecutive. Această problemă face parte din clasa de probleme de generare a **permutărilor cu condiție** – soluția conține o **condiție internă suplimentară** față de cea impusă de permutare. În acest exemplu, condiția suplimentară de continuare a construirii soluției este ca numărul ales pentru nivelul k al soluției să nu difere printr-o unitate de numărul care se găsește pe nivelul $k-1$ al soluției. Modificarea apare în subprogramul `valid()`:

```

int valid()
{
    if (k>1 && abs(s[k]-s[k-1])==1) return 0;
    for (int i=1; i<k; i++) if (s[i]==s[k]) return 0;
    return 1;
}

```

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor permutărilor.

1. Să se genereze toate permutările unei mulțimi de numere oarecare. Numerele se memorează într-un vector. **Indicație.** Se permută indicii elementelor din vectorul v – și vectorul soluție, s , conține indicii elementelor din vectorul v . În subprogramul `tipar()` se afișează elementele $v[s[i]]$.
2. Să se genereze toate funcțiile bijective $f: A \rightarrow B$, unde $\text{card}(A) = \text{card}(B) = n$.
3. Să se genereze toate posibilitățile de aranjare, pe o tablă de șah cu dimensiunea $n \times n$, a n ture care să nu se atace între ele. Deoarece tura se poate deplasa numai pe linia sau pe coloana pe care a fost plasată, turele se pot ataca între ele pe linie și pe coloană. **Indicație.** Se observă că fiecare tură trebuie să fie plasată singură pe o coloană, ca să nu se atace între ele. Soluția problemei este dată de mulțimea cu n elemente $\{x_1, x_2, \dots, x_n\}$, care se memorează în vector. Elementul soluției, x_k , reprezintă numărul liniei în care se așază tura din coloana k și se memorează pe nivelul k din vectorul s . Deoarece două ture nu pot fi așezate pe aceeași linie, vectorul trebuie să conțină elemente distincte. Problema se reduce la generarea permutărilor mulțimii $\{1, 2, 3, \dots, n\}$. Interpretarea soluției este: fiecare tură se plasează în coloana k , pe linia $s[k]$.
4. Să se genereze toate permutările mulțimii $\{1, 2, 3, \dots, n\}$, în care două numere vecine nu trebuie să fie ambele pare sau ambele impare. **Indicație.** În subprogramul `valid()` se mai verifică și condiția, suplimentară, de vecinătate.

```
int valid()
{if (k>1 && (s[k-1]%2==0 && s[k]%2==0) ||
    (s[k-1]%2==1 && s[k]%2==1)) return 0;
for (int i=1;i<k;i++)
    if (s[i]==s[k]) return 0;
return 1;}
```

5. Să se genereze toate permutările mulțimii $\{1, 3, 5, \dots, 2 \times n + 1\}$. **Indicație.** Soluția are n elemente. În subprogramul `init()` elementul de pe nivelul curent al soluției se inițializează cu valoarea -1 , iar în subprogramul `succesor()` se modifică modul de determinare a succesorului.

```
int succesor()
{if (s[k]<2*n+1) {s[k]= s[k]+2; return 1;}
else return 0;}
```

6. Să se genereze toate permutările unei mulțimi de numere oarecare, astfel încât cea mai mică și cea mai mare valoare să-și păstreze pozițiile inițiale.
7. Într-un șir sunt aranjate n persoane. Să se genereze toate posibilitățile de rearanjare în șir a acestor persoane, astfel încât fiecare persoană din șir:
- să nu aibă în fața sa aceeași persoană pe care a avut-o în șirul inițial;
 - să nu aibă aceiași vecini ca în șirul inițial;
 - să nu aibă în fața sa persoanele pe care le-a avut în șirul inițial.
 - să fie despărțită – de vecinii pe care i-a avut în șirul inițial – de una sau cel mult p persoane; valoarea pentru p se citește de la tastatură.

1.5.3.2. Generarea produsului cartezian

Se generează toate elementele produsului cartezian $A_1 \times A_2 \times A_3 \times \dots \times A_n$, unde $A_i = \{1, 2, \dots, n_i\}$.

O soluție a problemei va fi un element al produsului cartezian – și va fi formată din n elemente. Un element x_k al soluției poate fi orice element din mulțimea A_k (nu există **condiții interne**). Numărul de elemente ale fiecărei mulțimi A_i va fi memorat într-un vector m , cu lungimea logică n , unde $m[i] = n_i$. Față de algoritmul pentru generarea permutărilor, apar următoarele diferențe:

→ Deoarece fiecare mulțime A_k are un număr diferit de elemente, elementul k al soluției are succesor dacă numărul i de pe acest nivel este mai mic decât $m[k]$.

→ Deoarece nu există **condiții interne**, nu există restricții nici pentru **condiția de continuare** – și subprogramul `valid()` va furniza valoarea 1.

Implementarea iterativă

```
#include<iostream.h>
int n,k,ev,as,s[100];
void init()
{s[k]=0;}
int succesor()
{if (s[k]<m[k])
    {s[k]=s[k]+1; return 1;}
else return 0;}
int valid()
{return 1;}
int solutie()
{return k==n;}
```

Implementarea recursivă

```
#include<iostream.h>
int n,s[100];
void init(int k)
{s[k]=0;}
int succesor(int k)
{if (s[k]<m[k])
    {s[k]=s[k]+1; return 1;}
else return 0;}
int valid(int k)
{return 1;}
int solutie(int k)
{return k==n;}
```



```

void tipar()
{for(int i=1;i<=n;i++)
    cout<<s[i]<<" ";
  cout<<endl;}
void bt()
{
  k=1; init();
  while (k>0)
  {
    as=1; ev=0;
    while(as && !ev)
    {
      as=succesor();
      if(as) ev=valid();
    }
    if(as)
    {
      if (solutie()) tipar();
      else {k++; init();}
    }
    else k--;
  }
}
void main()
{
  cout<<"n= "; cin>>n;
  for(int i=1;i<=n;i++)
  {
    cout<<"Nr. de elemente multimea "
      <<i<<" ";
    cin>>m[i];
  }
  bt();
}

```

```

void tipar()
{for(int i=1;i<=n;i++)
    cout<<s[i]<<" ";
  cout<<endl;}
void bt(int k)
{
  init(k);
  while(succesor(k))
  {
    if(valid(k))
    {
      if(solutie(k)) tipar();
      else bt(k+1);
    }
  }
}
void main()
{
  cout<<"n= "; cin>>n;
  for(int i=1;i<=n;i++)
  {
    cout<<"Nr. de elemente multimea "
      <<i<<" ";
    cin>>m[i];
  }
  bt(1);
}

```

Algoritmul de generare a produsului cartezian poate fi folosit și în alte probleme. De exemplu, pentru **generarea tuturor submulțimilor unei mulțimi**.

O **submulțime** este formată din elemente ale mulțimii A . Pentru simplificarea algoritmului, vom considera mulțimea $A=\{1,2,3,\dots,n\}$. Considerăm mulțimea $B=\{0,1\}$. Pentru construirea submulțimilor, pentru fiecare submulțime se definește funcția $f:A \rightarrow B$, astfel: dacă elementul i din mulțimea A aparține submulțimii, atunci $f(i)=1$; altfel, $f(i)=0$. Problema se reduce la generarea produsului cartezian B^n . Soluția va fi memorată în vector și va avea n elemente, fiecare element k al soluției având valoarea 0 sau 1, cu semnificația că elementul k din mulțimea A aparține, respectiv nu aparține submulțimii. Alăturat sunt prezentate toate submulțimile mulțimii $\{1,2,3\}$ și conținutul vectorului soluție, pentru fiecare dintre ele.

Submulțimile	Vectorul
$\{\}=\emptyset$	0 0 0
$\{3\}$	0 0 1
$\{2\}$	0 1 0
$\{2, 3\}$	0 1 1
$\{1\}$	1 0 0
$\{1,3\}$	1 0 1
$\{1,2\}$	1 1 0
$\{1,2,3\}$	1 1 1

Față de programul anterior, apar următoarele modificări:

- Deoarece mulțimile produsului cartezian sunt identice ($B=\{0,1\}$), s-au modificat: subprogramul **init()** – inițializarea nivelului k al soluției se face cu valoarea -1, cu 1 mai mic decât 0 – și subprogramul **succesor()** – elementul are succesor numai dacă este mai mic decât ultimul element din mulțime, 1.
- În subprogramul de afișare a soluției, va fi afișat numărul nivelului i pentru care, în vector, se memorează valoarea 1.

Implementarea iterativă

```

#include<iostream.h>
int n,k,ev,as,s[100];
void init()

```

Implementarea recursivă

```

#include<iostream.h>
int n,s[100];
void init (int k)

```

```

{s[k]=-1;}
int succesor()
{if (s[k]<1)
    {s[k]=s[k]+1; return 1;}
    else return 0;}

int valid()
{return 1;}
int solutie()
{return k==n;}
void tipar ()
{int i,x=0; cout<<"{";
    for (i=1;i<=n;i++)
        if (s[i]==1)
            {cout<<i<<" "; x=1;}
    if (x) cout<<"\b";
    cout<<"}"<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
    bt();}

```

```

{s[k]=0;}
int succesor(int k)
{if (s[k]<s[k-1]+1)
    {s[k]=s[k]+1;return 1;}
    else return 0;}

int valid()
{return 1;}
int solutie(int k)
{return k==n;}
void tipar ()
{int i,x=0; cout<<"{";
    for (i=1;i<=n;i++)
        if (s[i]==1)
            {cout<<i<<" "; x=1;}
    if (x) cout<<"\b";
    cout<<"}"<<endl;}
cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
    bt(1);}

```

Observație. Caracterul **escape** '\b' este caracterul **Backspace** (șterge ultimul caracter din șir)

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea produsului cartezian.

1. Într-un restaurant, un meniu este format din trei feluri de mâncare. Există patru preparate culinare pentru felul unu, cinci preparate culinare pentru felul doi și trei preparate culinare pentru felul 3. Să se genereze toate meniurile care se pot forma cu aceste preparate culinare.
2. Pentru o mulțime oarecare **A**, cu **n** elemente, să se genereze toate submulțimile care au suma elementelor egală cu **s**. Numărul de elemente ale mulțimii, elementele mulțimii **A** și valoarea pentru suma **s** – se citesc de la tastatură.
3. Să se afișeze toate numerele cu **n** cifre ($1 \leq n \leq 10$) care au proprietatea că sunt formate numai din cifre pare, în ordine descrescătoare.
4. Să se afișeze toate numerele formate din cifre distincte, cu proprietatea că suma cifrelor este **S**. Valoarea variabilei **S** se citește de la tastatură.
5. Să se afișeze toate secvențele de **n** litere (**n** număr natural par, citit de la tastatură) din mulțimea {A,B,C,D}, secvențe care se construiesc astfel: nu se așează două litere identice una lângă alta – și trebuie să se folosească exact **n/2** litere A.
6. Să se rezolve, în mulțimea numerelor naturale, ecuația $4x+3y+2xy=48$. **Indicație.** Soluția are 2 elemente: **x** și **y**. Ele pot lua valori în intervalul [0,16]. Limita inferioară a intervalului este 0, pentru că numerele sunt naturale. Limita superioară s-a determinat considerând – pe rând – situațiile limită, $x=0$ și $y=0$. Considerăm mulțimea $A=\{0,2,3,\dots,16\}$. Problema se

reduce la generarea **produsului cartezian cu condiție $A \times A$** – soluția conține o **condiție internă suplimentară**: elementele ei trebuie să verifice ecuația.

7. Se citesc **n** cifre distincte și un număr natural **x**. Să se genereze toate numerele care se pot forma cu aceste cifre, și sunt mai mici decât numărul **x**. De exemplu, pentru cifrele 0, 1 și 3 și numărul **x=157**, se generează 1, 3, 10, 11, 13, 30, 31, 33, 100, 101, 103, 110, 111, 113, 130, 131, 133. **Indicație.** Se calculează **m**, numărul de cifre ale numărului **x**. Pentru mulțimea **A**, formată din cele **n** cifre, se generează produsele carteziane **A^p** , cu $1 \leq p \leq m$. Elementele produsului cartezian sunt cifrele numărului care se formează. Pentru ca un element al produsului cartezian să fie soluție, trebuie ca primul element să fie diferit de **0** (cifra cea mai semnificativă din număr nu trebuie să fie 0), iar numărul format cu **m** cifre – să fie mai mic decât numărul **x**.
8. Să se genereze toate numerele naturale, cu cel mult **n** cifre ($n \leq 10$), care sunt formate numai din cifre pare, în ordine strict crescătoare.
9. Să se genereze toate numerele naturale, cu **n** cifre, care conțin **p** cifre **k**. Valorile pentru **n**, **p** și **k** se citesc de la tastatură.
10. Se citește un număr natural **n**. Să se genereze toate numerele naturale care – reprezentate în baza 2 – au același număr de cifre de 0 și același număr de cifre de 1 ca și reprezentarea în baza 2 a numărului **n**.
11. Pe un bilet există 12 poziții care pot fi perforate, aranjate pe 4 linii și 3 coloane. Să se genereze toate posibilitățile de perforare a celor 12 poziții, astfel încât să nu existe două poziții alăturate neperforate. **Indicație.** Considerăm mulțimea **A**, formată din 12 elemente, fiecare element reprezentând o poziție care poate fi perforată, și mulțimea **B** = {0,1}. Se definește funcția **$f: A \rightarrow B$** , astfel: dacă poziția **i** este perforată, atunci **$f(i)=1$** ; altfel, **$f(i)=0$** . Problema se reduce la generarea **produsului cartezian B^{12}** – soluția conține o **condiție internă suplimentară**: poziția **k**, care se adaugă, nu trebuie să fie neperforată (nu trebuie să aibă valoarea 0) dacă poziția **k-1** sau poziția **k-3** este neperforată (are valoarea 0).

1.5.3.3. Generarea aranjamentelor

Se generează toate aranjamentele de **m** elemente ale mulțimii {1, 2, 3, ..., n}.

O soluție a problemei va fi un aranjament și va avea **m** elemente. Față de algoritmul pentru generarea permutărilor, se modifică doar condiția prin care se verifică dacă s-au obținut toate elementele soluției.

Implementarea iterativă

```
#include<iostream.h>
int n,m,k,ev,as,s[100];
void init()
{s[k]=0;}
int sucesor()
{if (s[k]<n)
    {s[k]=s[k]+1; return 1;}
    else return 0;}
int valid()
{for(int i=1;i<k;i++)
    if (s[k]==s[i]) return 0;
return 1;}
int solutie()
{return k==m;}
```

Implementarea recursivă

```
#include<iostream.h>
int n,m,s[100];
void init(int k)
{s[k]=0;}
int sucesor(int k)
{if (s[k]<n)
    {s[k]=s[k]+1; return 1;}
    else return 0;}
int valid(int k)
{for(int i=1;i<k;i++)
    if (s[k]==s[i]) return 0;
return 1;}
int solutie(int k)
{return k==m;}
```

```

void tipar()
{for(int i=1;i<=m;i++)
    cout<<s[i]<<" ";
  cout<<endl; }
void bt()
{ //partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
  cout<<"m= "; cin>>m;
  bt(); }

```

```

void tipar()
{for(int i=1;i<=m;i++)
    cout<<s[i]<<" ";
  cout<<endl; }
void bt(int k)
{ //partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
  cout<<"m= "; cin>>m;
  bt(1); }

```

Algoritmul de generare a aranjamentelor poate fi folosit și în alte probleme. De exemplu, pentru **generarea tuturor funcțiilor injective**.

Se generează toate funcțiile injective $f: A \rightarrow B$, unde $\text{card}(A)=m$ și $\text{card}(B)=n$. Pentru simplificarea algoritmului, vom considera mulțimile $A=\{1,2,3,\dots,m\}$ și $B=\{1,2,3,\dots,n\}$. O soluție este formată din m elemente. Elementul k al soluției reprezintă valoarea funcției: $f(k)$. Deoarece valoarea funcției $f(k)$ aparține mulțimii B , în vectorul soluție se vor genera numere din mulțimea $\{1,2,3,\dots,n\}$. Din definiția funcției injective, $f(i) \neq f(j)$, pentru orice $i \neq j$. Rezultă că, pentru ca funcția să fie injectivă, trebuie ca $m \leq n$. Problema se reduce la generarea tuturor aranjamentelor de n elemente luate câte m . Soluțiile vor fi afișate sub forma **tabelului de variație al funcției**. De exemplu, dacă $A=\{1,2\}$ și $B=\{1,2,3\}$, soluțiile și modul în care vor fi afișate sunt prezentate alăturat.

Față de programul anterior – nu se va modifica decât subprogramul de afișare a soluțiilor `tipar()`.

Soluția	Afișarea	
1 2	x	1 2
	f(x)	1 2
1 3	x	1 2
	f(x)	1 3
2 1	x	1 2
	f(x)	2 1
2 3	x	1 2
	f(x)	2 3
3 1	x	1 2
	f(x)	3 1
3 2	x	1 2
	f(x)	3 2

```

...
void tipar()
{int i; cout<<" x | ";
  for (i=1;i<=m;i++) cout<<i<<" ";    cout<<endl;
  for (i=1;i<=m;i++) cout<<"-----";  cout<<endl<<"f(x) | ";
  for (i=1;i<=m;i++) cout<<s[i]<<" ";  cout<<endl<<endl; }
...
void main()
{cout<<"numarul de elemente ale multimii A= "; cin>>m;
  cout<<"numarul de elemente ale multimii B= "; cin>>n; bt(); }

```

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor aranjamentelor.

1. Să se genereze toate posibilitățile de aranjare pe m scaune a n persoane ($m \leq n$). Valorile pentru n și m și numele persoanelor se citesc dintr-un fișier text.
2. Se citesc de la tastatură n cifre distincte. Să se genereze toate numerele de m cifre ($m \leq n$) care se pot forma cu aceste cifre și care conțin toate cele n cifre.
3. Să se genereze toate anagramele unui cuvânt, din care se elimină p litere oarecare. Valoarea minimă a numărului p este 1, iar valoarea maximă este cu 1 mai mică decât lungimea cuvântului. Se citesc de la tastatură literele cuvântului și valoarea numărului p .

4. Se citesc de la tastatură n caractere distincte. Să se genereze toate cuvintele de m litere ($m \leq n$) care se pot forma cu aceste caractere și care conțin toate cele n caractere.
5. Se citește un număr n care are m cifre. Să se genereze toate numerele, cu cel mult m cifre, care se pot forma cu cifrele numărului inițial.
6. Dintr-o mulțime de n persoane, aranjate într-un șir, se elimină p persoane. Să se genereze toate posibilitățile de aranjare într-un șir a persoanelor rămase, astfel încât fiecare persoană să nu aibă aceiași vecini ca în șirul inițial. Valorile pentru n și p se citesc de la tastatură.
7. Să se genereze toate aranjamentele de m numere ale unei mulțimi de n numere oarecare, astfel încât suma elementelor generate să fie egală cu S . Numărul de elemente ale mulțimii, n , elementele mulțimii, valorile pentru m și pentru suma S – se citesc de la tastatură.
8. Să se genereze toate drapelele cu trei culori care se pot forma cu șase culori: alb, galben, roșu, verde, albastru și negru – care au în mijloc culoarea alb, verde sau roșu.
Indicație. Soluția are 3 elemente, un element al soluției fiind indicele din vector al unei culori. Se generează **aranjamente cu condiție** de 6 obiecte luate câte 3 – soluția conține o **condiție internă suplimentară** față de cea impusă de aranjamente: elementul 2 al soluției trebuie să fie indicele culorii alb, verde sau roșu.
9. Se consideră n cuburi. Fiecare cub i are latura L_i și culoarea c_i . Să se construiască toate turnurile stabile formate cu m cuburi, astfel încât două cuburi vecine în turn să nu aibă aceeași culoare. Valorile pentru n și m și atributele celor n cuburi se citesc de la tastatură.
Indicație. Informațiile despre cuburi se memorează în doi vectori (unul pentru lungimile laturilor, iar celălalt pentru culori) cu n elemente. Soluția are m elemente. Un element al soluției este indicele din vector al unui cub. Se generează **aranjamente cu condiție** de n obiecte luate câte m – soluția conține o **condiție internă suplimentară** față de cea impusă de aranjamente: cubul k , care se adaugă la turn, trebuie să aibă latura mai mică decât a cubului $k-1$ – și culoarea diferită de acesta.

1.5.3.4. Generarea combinărilor

Se generează toate combinările de m elemente ale mulțimii $\{1, 2, 3, \dots, n\}$.

O soluție a problemei va fi o combinație și va avea m elemente. Față de algoritmul pentru generarea aranjamentelor, apare o condiție suplimentară: aceea ca soluțiile obținute să fie distincte, adică două soluții să nu conțină aceleași numere. Pentru aceasta, se va adăuga o **condiție de continuare** suplimentară: valoarea de pe nivelul k trebuie să fie strict mai mare decât oricare dintre valorile de pe nivelele inferioare. Altfel spus, elementele soluției trebuie să fie ordonate: $s[1] < s[2] < \dots < s[k-1] < s[k]$. Condiția de continuare este îndeplinită dacă elementul de pe nivelul k va avea o valoare strict mai mare decât valoarea elementului de pe nivelul $k-1$ (se inițializează cu o valoare egală cu cea a elementului de pe nivelul $k-1$) și o valoare mai mică decât $n-m+k$ (se caută succesorul până la valoarea $n-m+k$).

Implementarea iterativă

```
#include<iostream.h>
int n,m,k,ev,as,s[100];
void init()
{if (k==1) s[k]=0;
  else s[k]=s[k-1];}
int succesor()
{if (s[k]<n-m+k)
 {s[k]=s[k]+1; return 1;}
  else return 0;}
```

Implementarea recursivă

```
#include<iostream.h>
int n,m,s[100];
void init(int k)
{ if (k==1) s[k]=0;
  else s[k]=s[k-1];}
int succesor(int k)
{if (s[k]<n-m+k)
 {s[k]=s[k]+1; return 1;}
  else return 0;}
```

```

int valid()
{return 1;}
int solutie()
{return k==m;}
void tipar()
{for(int i=1;i<=m;i++)
    cout<<s[i]<<" ";
    cout<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
    cout<<"m= "; cin>>m; bt();}

```

```

int valid()
{return 1;}
int solutie(int k)
{return k==m;}
void tipar()
{for(int i=1;i<=m;i++)
    cout<<s[i]<<" ";
    cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
    cout<<"m= "; cin>>m;
    bt(1);}

```

Algoritmul de generare a combinațiilor poate fi folosit în problemele de **generare a tuturor posibilităților de a forma – din n obiecte – grupuri de m obiecte, care să aibă anumite proprietăți**. De exemplu, din n obiecte trebuie să se distribuie unei persoane m obiecte, care să conțină obligatoriu obiectul p și să nu conțină obiectul q . Să se genereze toate posibilitățile de a forma grupuri de m astfel de obiecte. Pentru simplificare, vom considera mulțimea obiectelor ca fiind $\{1, 2, 3, \dots, n\}$. Valorile pentru numărul de obiecte, n , numărul de obiecte din grup, m , și indicii obiectelor p și q – se citesc de la tastatură.

Deoarece obiectul p face parte obligatoriu din grupul de obiecte, el va fi memorat ca prim element al soluției ($s[1]=p$), iar subprogramul $bt()$ va genera numai următoarele $m-1$ elemente ale soluției. Se modifică și condiția de continuare a construirii soluției (subprogramul $valid()$), deoarece obiectele p și q nu pot fi elemente ale soluției.

```

#include<iostream.h>
int n,m,p,q,k,ev,as,s[100];
void init() { //la fel ca în exemplul anterior}
int sucesor() { //la fel ca în exemplul anterior}
int valid()
{return s[k]!=p && s[k]!=q;}
int solutie() { //la fel ca în exemplul anterior}
void tipar() { //la fel ca în exemplul anterior}
void bt()
{k=2;init();
    while (k>1)
//la fel ca în exemplul anterior}
void main()
{cout<<"n= "; cin>>n; cout<<"m= "; cin>>m;
    cout<<"p= "; cin>>p; cout<<"q= "; cin>>q;
    s[1]=p; bt();}

```

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor combinațiilor.

1. Să se genereze toate grupurile, de p persoane, care se pot forma din n persoane. Informațiile se citesc dintr-un fișier text unde, pe primul rând, sunt scrise valorile pentru p și n , despărțite printr-un spațiu, iar pe următoarele rânduri, numele persoanelor, câte un nume pe un rând.

2. Două persoane își împart n obiecte, astfel: prima persoană ia m obiecte, iar cealaltă persoană – restul obiectelor. Să se genereze toate posibilitățile de distribuire a celor n obiecte între cele două persoane.
3. Două persoane își împart n obiecte. Fiecare obiect are o valoare v_i . Să se genereze toate posibilitățile de distribuire a celor n obiecte, între cele două persoane, astfel încât fiecare persoană să primească obiecte care să aibă aceeași valoare totală.
4. Să se genereze toate numerele binare de n cifre care au m cifre de 0. **Indicație.** O soluție este formată din m elemente, un element fiind poziția din număr în care se poate găsi cifra 0 – exceptând prima poziție.
5. Din n obiecte date, să se genereze toate grupurile de m obiecte care îndeplinesc următoarele condiții:
 - a. conțin exact p obiecte precizate;
 - b. nu conțin nici unul din p obiecte precizate;
 - c. conțin numai un obiect din p obiecte precizate;
 - d. conțin cel puțin un obiect din p obiecte precizate;
 - e. conțin exact q obiecte din p obiecte precizate;
 - f. conțin exact q obiecte din p obiecte precizate – și nu conțin r obiecte precizate;
 - g. conțin exact q obiecte din p obiecte precizate – și nu conțin s obiecte, din r obiecte precizate.

1.5.3.5. Generarea tuturor partițiilor unui număr natural

O partiție a unui număr natural nenul, n , este o descompunere a numărului n în sumă de m numere naturale nenule. Soluțiile care nu diferă decât prin ordinea termenilor nu sunt considerate distincte. Alături, sunt prezentate toate partițiile numărului 4.

$4=1+1+1+1$	$(m=4)$
$4=1+1+2$	$(m=3)$
$4=1+3$	$(m=2)$
$4=2+2$	$(m=2)$
$4=4$	$(m=1)$

O soluție a problemei va fi formată din m elemente, a căror sumă este egală cu numărul n . Elementele soluției reprezintă termenii unei descompuneri. Numărul de elemente ale unei soluții (m) este egal cu valoarea indicelui k atunci când s-a obținut soluția. Soluția se obține atunci când suma elementelor din vector este egală cu n . Altfel spus, soluția se obține atunci când suma $S=s[1]+s[2]+ \dots +s[k-1]+s[k]$ are valoarea n . Pentru a evita repetarea aceleiași descompuneri, valoarea de pe nivelul k trebuie să fie mai mare sau egală cu oricare dintre valorile de pe nivelele inferioare. Altfel spus, elementele soluției trebuie să fie ordonate: $s[1] \leq s[2] \leq \dots \leq s[k-1] \leq s[k]$. Această condiție este îndeplinită dacă elementul de pe nivelul k va avea o valoare mai mare sau egală cu valoarea elementului de pe nivelul $k-1$ (se inițializează cu o valoare mai mică cu 1 decât cea a elementului de pe nivelul $k-1$) și o valoare mai mică decât diferența dintre numărul n și suma termenilor descompunerii găsiți până la nivelul k , $S=s[1]+s[2]+ \dots +s[k-1]$.

Condiția de continuare este ca suma termenilor generați, S , să fie mai mică sau egală cu n ($S \leq n$), o soluție completă obținându-se atunci când $S=n$. Inițial, suma S are valoarea 0, și ea trebuie actualizată în permanență. Există două cazuri de actualizare:

→ Atunci când se adaugă la soluție un nou termen, acesta se adaugă și la sumă. Altfel spus, dacă succesorul găsit poate fi element al soluției (prin adăugarea lui la sumă, aceasta nu va depăși valoarea numărului n), atunci el se adaugă la sumă. Actualizarea sumei se va face în subprogramul `valid()` (termenul se adaugă la sumă numai dacă face parte din descompunere): $S+=s[k]$.

→ Atunci când se coboară la nivelul anterior al soluției, trebuie scăzute din sumă două valori: valoarea elementului de pe nivelul k și valoarea elementului de pe nivelul precedent ($k-1$). Deoarece în soluție se coboară întotdeauna după ce s-a obținut o soluție completă – și nu se mai poate găsi un element pentru nivelul k , cu care să se continue dezvoltarea soluției – scăderea din sumă a termenului de pe nivelul curent se va face după ce se afișează o soluție completă, în subprogramul `tipar()`: $S -= s[k]$, iar scăderea din sumă a termenului precedent se face în subprogramul `succesor()`, atunci când nu se găsește succesor pentru elementul de pe nivelul k : $S -= s[k-1]$.

Implementarea iterativă

```
#include<iostream.h>
int n,S=0,k,ev,as,s[100];
void init ()
{if (k==1) s[k]=0;
  else s[k]=s[k-1]-1;}
int sucesor()
{if (s[k]<n-S)
  {s[k]=s[k]+1; return 1;}
  else {S-=s[k-1]; return 0;}}
int valid()
{if (S+s[k]<=n)
  {S+=s[k]; return 1;}
  else return 0;}
int solutie()
{return S==n;}
void tipar()
{for(int i=1;<k;i++)
  cout<<s[i]<<" ";
  cout<<s[k]<<endl;
  S-=s[k];}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt();}
```

Implementarea recursivă

```
#include<iostream.h>
int n,S=0,s[100];
void init (int k)
{if (k==1) s[k]=0;
  else s[k]=s[k-1]-1;}
int sucesor(int k)
{if (s[k]<n-S)
  {s[k]=s[k]+1; return 1;}
  else {S-=s[k-1]; return 0;}}
int valid(int k)
{if (S+s[k]<=n)
  {S+=s[k]; return 1;}
  else return 0;}
int solutie(int k)
{return S==n;}
void tipar(int k)
{for(int i=1;<k;i++)
  cout<<s[i]<<" ";
  cout<<s[k]<<endl;
  S-=s[k];}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n;
  bt(1);}
```

Algoritmul de generare a tuturor partițiilor unui număr natural poate fi folosit și în alte probleme. De exemplu, **generarea tuturor posibilităților de plată a unei sume cu bancnote de valori date**.

Problema se reduce la generarea tuturor partițiilor unui număr natural nenul, **suma**, o partiție fiind o descompunere a numărului **suma** în sumă de **m** numere naturale nenule, cu valori aparținând mulțimii $A=\{a_1, a_2, \dots, a_n\}$. Alăturat, sunt prezentate toate partițiile sumei 25, pentru bancnote cu valori aparținând mulțimii $A=\{5, 10\}$.

25 = 5+5+5+5+5 (m=5)
25 = 5+5+5+10 (m=4)
25 = 5+10+10 (m=3)

Față de problema anterioară, apar următoarele modificări:

→ Deoarece suma **suma** se descompune în valori precizate, aparținând mulțimii **A**, aceste valori se memorează în vectorul **a**, care are lungimea logică **n** (numărul de valori ale bancnotelor).

- Un element al soluției este indicele valorii bancnotei din vectorul **a**, adică un element din mulțimea **{1,2, ..., n}**.
- O soluție a problemei va fi formată din **m** elemente, alese astfel încât suma valorilor bancnotelor corespunzătoare lor să fie egală cu numărul **suma**. Altfel spus, soluția se obține atunci când suma **S=a[s[1]]+a[s[2]]+ ... + a[s[k-1]]+ a[s[k]]** are valoarea **suma**.
- **Condiția de continuare** este ca suma valorilor termenilor generați, **S**, să fie mai mică sau egală cu **suma** (**S≤suma**), o soluție completă obținându-se atunci când **S=suma**. Atunci când se adaugă la soluție un nou element, se adună la sumă valoarea bancnotei cu indicele **k**: **S+=a[s[k]]**, iar când se coboară la nivelul anterior al soluției, se scade din sumă valoarea bancnotei cu indicele **k**: **S-=a[s[k]]** și valoarea bancnotei cu indicele **k-1**: **S+=a[s[k-1]]**.
- Deoarece este posibil ca, pentru anumite valori ale sumei **suma** și ale valorilor bancnotelor, să nu existe nici o soluție, se folosește variabila de tip logic **este**, care are valoarea 1 (*true*), dacă există cel puțin o soluție, și 0 (*false*) în caz contrar.

```
#include<iostream.h>
int n,k,ev,as,S=0,suma,este=0,a[10], s[100];
void init ()
{if (k==1) s[k]=0;
  else s[k]=s[k-1]-1;}
int sucesor()
{if (s[k]<n) {s[k]=s[k]+1; return 1;}
  else {S-=a[s[k-1]]; return 0;}}
int valid()
{if (S+a[s[k]]<=suma) {S+=a[s[k]]; return 1;}
  else return 0;}
int solutie()
{return S==suma;}
void tipar()
{int i,j,p,este=1;
  for (i=1;i<=n;i++)
    {for(j=1,p=0;j<=k;j++) if (i==s[j]) p++;
      if (p!=0) cout<<p<<"* "<<a[i]<<" + ";}
    cout<<'\\b'<<'\\b'<<" "<<endl; s-=a[s[k]];}
void bt() //partea fixă a algoritmului
{ ... }
void main()
{cout<<"n= "; cin>>n;
  for (int i=1;i<=n;i++)
    {cout<<"Valoare bancnota "<<i<<": "; cin>>a[i];}
  cout<<"suma= "; cin>>suma; bt();
  if (!este) cout<<"Imposibil"; }
```

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor partițiilor unui număr natural.

1. Să se genereze toate descompunerile unui număr natural, **n**, în numere naturale distincte.
2. Să se genereze toate descompunerile unui număr natural **n** în numere prime.
3. Să se genereze toate descompunerile unui număr natural **n** în sumă de 3 și 5.

4. O bară are lungimea L . Se consideră n repere de lungimi diferite. Să se genereze toate posibilitățile de a tăia bara după reperele existente, fără să rămână rest la tăiere, un reper putând fi folosit de mai multe ori. Se citește dintr-un fișier text, de pe primul rând, lungimea barei L și numărul de repere n , și de pe următorul rând – reperele. Numerele de pe un rând sunt separate prin spațiu.
5. Să se găsească modalitatea de plată a unei sume – cu un număr minim de bancnote cu valori date.

1.5.3.6. Generarea tuturor partițiilor unei mulțimi

O partiție a unei mulțimi A este formată din mulțimile nevide disjuncte A_i , a căror reuniune

este mulțimea A : $\bigcup_{i=1}^m A_i = A$ și $A_i \cap A_j = \Phi$, pentru orice $i, j = 1 \div m$.

Pentru simplificarea algoritmului vom considera mulțimea $A = \{1, 2, 3, \dots, n\}$. O partiție va fi formată din m mulțimi, cu $1 \leq m \leq n$. Soluția va fi memorată în vector și va avea n elemente, fiecare element k al soluției reprezentând mulțimea i ($1 \leq i \leq n$) căreia îi aparține elementul k din mulțimea care se partiționează: $s[k] = i$ înseamnă că elementul k din mulțimea A face parte din mulțimea i a partiției. În cadrul unei partiții nu interesează ordinea în care apar elementele mulțimii A . Cel mai mare număr care va fi generat în vector reprezintă numărul de mulțimi m în care a fost descompusă mulțimea A . În plus, numerele generate trebuie să aparțină unei mulțimi de numere consecutive care începe cu 1, deoarece partiția nu conține mulțimi vide. Alături sunt prezentate toate partițiile mulțimii $\{1, 2, 3\}$ și conținutul vectorului pentru fiecare dintre ele. În exemplu, nu există soluția 1 3 3, deoarece aceste valori nu aparțin unei mulțimi de numere consecutive (nu se poate ca un element din mulțimea A să aparțină mulțimii A_1 și alte două elemente mulțimii A_3 , deoarece ar însemna că mulțimea A_2 este vidă).

Partițiile	Vectorul
$\{1, 2, 3\}$	1 1 1
$\{1, 2\} \{3\}$	1 1 2
$\{1, 3\} \{2\}$	1 2 1
$\{1\} \{2, 3\}$	1 2 2
$\{1\} \{2\} \{3\}$	1 2 3

Condiția de continuare este asigurată prin modul în care este ales succesorul $s[k] \leq k$, ceea ce înseamnă că elementul k din mulțimea A nu poate aparține decât unei partiții al cărei număr este mai mic sau cel mult egal cu numărul de ordine al elementului. Altfel spus, dacă până la nivelul k numărul maxim atribuit pentru o partiție este i , acest număr reprezentând numărul de mulțimi ale partiției care există în soluția parțială, pe nivelul $k+1$ elementul poate avea una dintre următoarele valori:

- Orice valoare de la 1 la i , ceea ce înseamnă că elementul $k+1$ din mulțimea A se adaugă la una dintre mulțimile care există deja.
- Valoarea $i+1$, ceea ce înseamnă că elementul $k+1$ din mulțimea A va genera o nouă mulțime în partiție.

Implementarea iterativă

```
#include<iostream.h>
int n,k,ev,as,s[100];
void init()
{s[k]=0;}
int succes()
{if (s[k]<s[k-1]+1)
    {s[k]=s[k-1]+1; return 1;}
 else return 0;}
```

Implementarea recursivă

```
#include<iostream.h>
int n,s[100];
void init (int k)
{s[k]=0;}
int succes(int k)
{if (s[k]<s[k-1]+1)
    {s[k]=s[k-1]+1;return 1;}
 else return 0;}
```

```

int valid()
{return 1;}
int solutie()
{return k==n;}
void tipar()
{int i,j,max=st[1];
 for(i=2;i<=n;i++)
  if (s[i]>max) max=s[i];
 for (i=1;i<=max;i++)
  {cout<<"{";
   for (j=1;j<=n;j++)
    if (s[j]==i) cout<<j<<" ";
   cout<<"\b"<<" ";}
cout<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt();}

```

```

int valid()
{return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{int i,j,max=s[1];
 for(i=2;i<=n;i++)
  if (s[i]>max) max=s[i];
 for (i=1;i<=max;i++)
  {cout<<"{";
   for (j=1;j<=n;j++)
    if (s[j]==i) cout<<j<<" ";
   cout<<"\b"<<" ";}
cout<<endl;}
void bt(int k)
{//partea fixă a algoritmului}
void main()
{cout<<"n= "; cin>>n; bt(1);}

```

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor partițiilor unei mulțimi. Se consideră mulțimea **A**, cu **n** numere întregi. Valorile pentru **n** și **m** și cele pentru elementele mulțimii **A** se citesc de la tastatură.

1. Să se genereze toate partițiile mulțimii **A** formate din două submulțimi, care au suma elementelor egale.
2. Să se genereze toate partițiile mulțimii **A** formate din **m** submulțimi.
3. Să se genereze toate partițiile mulțimii **A** formate din submulțimi care au același număr de elemente.

1.5.3.7. Generarea tuturor funcțiilor surjective

Se generează toate funcțiile surjective $f: A \rightarrow B$, unde $\text{card}(A)=m$ și $\text{card}(B)=n$. Pentru simplificarea algoritmului vom considera mulțimile $A=\{1,2,3,\dots,m\}$ și $B=\{1,2,3,\dots,n\}$. O soluție este formată din **m** elemente. Elementul **k** al soluției reprezintă valoarea funcției: **f(k)**. Deoarece valoarea funcției **f(k)** aparține mulțimii **B**, în vector se vor genera numere din mulțimea $\{1,2,3,\dots,n\}$. Din definiția funcției surjective, trebuie ca, pentru orice $j \in B$, să existe $i \in A$, astfel încât **f(i)=j**. Rezultă că, pentru ca funcția să fie surjectivă, trebuie ca $n \leq m$. Problema se reduce la generarea în vectorul soluție a tuturor elementelor produsului cartezian B^m , din care vor fi considerate soluții numai cele care conțin toate elementele din mulțimea **B**. Soluțiile vor fi afișate sub forma **tabelului de variație al funcției**. De exemplu, dacă $A=\{1,2,3\}$ și $B=\{1,2\}$, soluțiile și modul în care vor fi afișate sunt prezentate alăturat.

Soluția	Afișarea				
1 1 2	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>1 1 2</td></tr> </table>	x	1 2 3	f(x)	1 1 2
x	1 2 3				
f(x)	1 1 2				
1 2 1	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>1 2 1</td></tr> </table>	x	1 2 3	f(x)	1 2 1
x	1 2 3				
f(x)	1 2 1				
1 2 2	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>1 2 2</td></tr> </table>	x	1 2 3	f(x)	1 2 2
x	1 2 3				
f(x)	1 2 2				
2 1 1	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>2 1 1</td></tr> </table>	x	1 2 3	f(x)	2 1 1
x	1 2 3				
f(x)	2 1 1				
2 2 1	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>2 2 1</td></tr> </table>	x	1 2 3	f(x)	2 2 1
x	1 2 3				
f(x)	2 2 1				
2 1 2	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>2 1 2</td></tr> </table>	x	1 2 3	f(x)	2 1 2
x	1 2 3				
f(x)	2 1 2				
2 2 1	<table> <tr><td>x</td><td>1 2 3</td></tr> <tr><td>f(x)</td><td>2 2 1</td></tr> </table>	x	1 2 3	f(x)	2 2 1
x	1 2 3				
f(x)	2 2 1				

Condiția de continuare este asigurată prin modul în care este ales succesul ca element din mulțimea **B**, singurul caz special fiind al elementului care se adaugă pe ultimul nivel din vectorul soluție (**m**). Prin adăugarea acestui element trebuie ca în

vector să existe toate elementele mulțimii **B**. Această condiție este verificată prin funcția **surjectiva()** care furnizează un rezultat logic: 1 (*true*), dacă în vector există toate cele **n** elemente ale mulțimii **B**, și 0 (*false*) în caz contrar.

Implementarea iterativă

```
#include<iostream.h>
int n,m,k,ev,as,s[100];
int surjectiva()
{int i,j,x;
  for (j=1;j<=n;j++)
    {for (i=1,x=0;i<=m && !x;i++)
      if (st[i]==j) x=1;
      if (!x) return 0;}
return 1;}
void init() {s[k]=0;}
int sucesor()
{if (s[k]<s[k-1]+1)
  {s[k]=s[k-1]+1; return 1;}
  else return 0;}
int valid()
{if (k==m)
  if (!surjectiva()) return 0;
return 1;}
int solutie()
{return k==m;}
void tipar()
{int i; cout<<" x | ";
  for (i=1;i<=m;i++) cout<<i<<" ";
  cout<<endl;
  for (i=1;i<=m;i++) cout<<"-----";
  cout<<endl<<"f(x) | ";
  for (i=1;i<=m;i++) cout<<s[i]<<" ";
  cout<<endl<<endl;}
void bt()
{//partea fixă a algoritmului}
void main()
{cout<<"elemente multimea A= ";
cin>>m;
cout<<"elemente multimea B= ";
cin>>n; bt();}
```

Implementarea recursivă

```
#include<iostream.h>
int n,m,s[100];
int surjectiva()
{int i,j,x;
  for (j=1;j<=n;j++)
    {for (i=1,x=0;i<=m && !x;i++)
      if (s[i]==j) x=1;
      if (!x) return 0;}
void init (int k)
{{s[k]=0;}
int sucesor(int k)
{if (s[k]<s[k-1]+1)
  {s[k]=s[k-1]+1;return 1;}
  else return 0;}
int valid(int k)
{if (k==m)
  if (!surjectiva()) return 0;
return 1;}
int solutie(int k)
{return k==n;}
void tipar()
{ int i; cout<<" x | ";
  for (i=1;i<=m;i++) cout<<i<<" ";
  cout<<endl;
  for (i=1;i<=m;i++) cout<<"-----";
  cout<<endl<<"f(x) | ";
  for (i=1;i<=m;i++) cout<<s[i]<<" ";
  cout<<endl<<endl;}
void bt (int k)
{//partea fixă a algoritmului}
void main()
{cout<<"elemente multimea A= ";
cin>>m;
cout<<"elemente multimea B= ";
cin>>n; bt(1);}
```

Temă



Scrieți următoarele programe, în care să folosiți metoda backtracking pentru generarea tuturor funcțiilor surjective.

1. Se citesc de la tastatură **n** cifre distincte. Să se genereze toate numerele de **m** cifre ($n \leq m$) care se pot forma cu aceste cifre și care conțin toate cele **n** cifre.
2. Se citesc de la tastatură **n** caractere distincte. Să se genereze toate cuvintele de **m** caractere ($n \leq m$) care se pot forma cu aceste caractere și care conțin toate cele **n** caractere.

3. Profesorul de informatică a pregătit m teme pentru proiecte, pe care trebuie să le repartizeze celor n elevi din clasă ($m \leq n$), astfel încât nici o temă de proiect să nu rămână nerepartizată. Să se genereze toate soluțiile de repartizare a temelor pentru proiect.
4. Să se genereze toate construcțiile corecte de n paranteze (și) – n fiind un număr par citit de la tastatură. De exemplu, pentru $n=6$ construcțiile $(())()$ și $()(())$ sunt corecte, iar construcția $))(($ nu este corectă. **Indicație.** Se generează funcțiile surjective $f: A \rightarrow B$, unde $A = \{1, 2, 3, \dots, n\}$ = mulțimea pozițiilor ocupate de paranteze, și $B = \{0, 1\}$ = mulțimea parantezelor – ($=0$ și)= 1 – care îndeplinesc următoarele condiții:
- $f(1)=0$ și $f(n)=1$ (expresia începe cu paranteză deschisă și se termină cu o paranteză închisă).
 - Numărul de valori 0 ale funcției și numărul de valori 1 ale funcției sunt egale cu $n/2$ (numărul de paranteze deschise este egal cu numărul de paranteze închise).
 - În timpul construirii soluției nu trebuie ca numărul de valori 1 ale funcției să fie mai mare decât numărul de valori 0 generate până la acel moment (numărul de paranteze închise este întotdeauna cel mult egal cu numărul de paranteze deschise).

1.5.3.8. Problema celor n dame

Se generează toate posibilitățile de aranjare, pe o tablă de șah, cu dimensiunea $n \times n$, a n dame, care să nu se atace între ele. Damele se pot ataca între ele pe linie, pe coloană și pe diagonală.

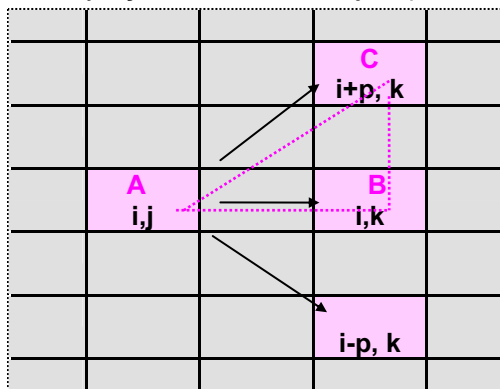
Se observă că fiecare damă trebuie să fie plasată singură pe o coloană, ca să nu se atace între ele. Soluția problemei este dată de mulțimea cu n elemente $\{x_1, x_2, \dots, x_n\}$, care se memorează în vector. Elementul soluției, x_k , reprezintă numărul liniei în care se așază dama din coloana k și se memorează pe nivelul k al vectorului s . De exemplu, pentru $n=8$, o soluție este $S=\{4, 8, 1, 5, 7, 2, 6, 3\}$ – și damele sunt aranjate pe tabla de șah ca în figura alăturată.

8		⊙						
7					⊙			
6							⊙	
5				⊙				
4	⊙							
3								⊙
2						⊙		
1			⊙					
	1	2	3	4	5	6	7	8

Date fiind coordonatele i și j de pe tabla de șah ale poziției unei dame care a fost așezată anterior (linia i este memorată pe nivelul j în vector – $i = st[j]$), ca ea să nu atace dama care urmează să fie pusă în coloana k trebuie să fie îndeplinite următoarele **condiții interne**:

→ Dama din coloana k nu trebuie să se găsească pe linia i .

→ Dama din coloana k nu trebuie să se găsească pe diagonala cu dama din coloana j , adică triunghiul dreptunghic ABC nu trebuie să fie isoscel. Pentru ca triunghiul ABC să nu fie isoscel, condiția pe care trebuie să o îndeplinească dama din coloana k se poate exprima astfel: oricare ar fi $j < k$, $x_j \neq x_k$ și $|x_k - x_j| \neq k - j$.



Aceste condiții interne ale soluției trebuie să se regăsească în **condiția de continuare a soluției**, care se verifică în subprogramul `valid()`: $s[j] \neq s[k]$ și $abs(s[k] - s[j]) \neq k - j$ pentru orice $j \neq k$.

```
#include<iostream.h>
#include<math.h>
int n,k,ev,as,s=0,suma,este=0,a[10],s[100];
void init() {s[k]=0;}
int sucesor()
{if (s[k]<n) {s[k]=s[k]+1; return 1;}
  else return 0;}
int valid()
{for(int i=1;i<k;i++)
  if (s[k]==s[i] || abs(s[k]-s[i])==k-i) return 0;
  return 1;}
int solutie()
{return k==n;}
void tipar()
{for(int i=1;i<=n;i++) cout<<s[i]<<" "; cout<<endl;}
void bt() { ... } //partea fixa a algoritmului
void main()
{cout<<"n= "; cin>>n; bt();}
```

Temă



Să se genereze toate posibilitățile de aranjare, pe o tablă de șah, cu dimensiunea $n \times n$, a n nebuni, care să nu se atace între ei. Nebunul se poate deplasa numai pe diagonală.

1.6. Metoda „Divide et Impera“

1.6.1. Descrierea metodei „Divide et Impera“

Metoda **Divide et Impera** se poate folosi pentru problemele care **pot fi descompuse în subprobleme similare cu problema inițială** (care se rezolvă prin aceeași metodă) și care prelucrează mulțimi de date de dimensiuni mai mici, **independente unele de altele** (care folosesc **mulțimi de date de intrare disjuncte**).

Studiu de caz

Scop: identificarea problemelor care **pot fi descompuse în subprobleme similare** care folosesc **mulțimi de date de intrare disjuncte**.

Enunțul problemei 1: Să se calculeze suma elementelor dintr-un vector v care conține numere întregi.

Mulțimea datelor de intrare o reprezintă cele n elemente ale vectorului v . Ele pot fi divizate în câte două submulțimi disjuncte, prin divizarea mulțimii indicilor în două submulțimi. Mulțimea inițială a indicilor este determinată de primul indice (s) și de ultimul indice (d), iar intervalul indicilor care se divizează este $[s,d]$. El se divizează în două submulțimi disjuncte, $[s,m]$ și $[m+1,d]$, unde m este indicele din mijlocul intervalului: $m=(s+d)/2$. Astfel, problema inițială este descompusă în două subprobleme, fiecare dintre ele constând în calcularea sumei numerelor dintr-o submulțime de elemente (care corespunde unui subinterval al indicilor). Descompunerea continuă până când fiecare submulțime conține un singur element – și se poate calcula suma, obținându-se soluția subproblemei.

Enunțul problemei 2: Să se calculeze suma $1 \times 2 + 2 \times 3 + \dots + n \times (n+1)$.

Mulțimea datelor de intrare o reprezintă primele n numere naturale. Mulțimea inițială este determinată de primul număr ($s=1$) și de ultimul număr ($d=n$), iar intervalul care se divizează este $[s,d]$. El se divizează în două submulțimi disjuncte, $[s,m]$ și $[m+1,d]$, unde m este numărul din mijlocul intervalului: $m=(s+d)/2$. Astfel, problema inițială este descompusă în două subprobleme, fiecare dintre ele constând în calcularea sumei produselor dintre două numere consecutive dintr-o submulțime de elemente (care corespunde unui subinterval al numerelor). Descompunerea continuă până când fiecare submulțime conține un singur element – și se poate calcula produsul, care se va adăuga la sumă, pentru a obține soluția subproblemei.

Enunțul problemei 3: Să se genereze termenul n al șirului lui **Fibonacci**.

Șirul lui Fibonacci este definit recursiv: $f_1=1$, $f_2=1$ și $f_n = f_{n-2} + f_{n-1}$, pentru $n \geq 3$. Problema determinării termenului n al șirului lui Fibonacci se poate descompune în două subprobleme: determinarea termenului $n-1$ și determinarea termenului $n-2$. Descompunerea continuă până când trebuie determinați termenii f_1 și f_2 , a căror valoare este cunoscută.



Metoda „Divide et Impera” se bazează pe descompunerea unei probleme în subprobleme similare, prin intermediul unui proces recursiv. Procesul recursiv de descompunere a unei subprobleme în alte subprobleme continuă până se obține o subproblemă cu rezolvarea imediată (cazul de bază), după care se compun soluțiile subproblemelor – până se obține soluția problemei inițiale.

Pașii algoritmului sunt:

- PAS1.** Se **descompune** problema în subprobleme similare problemei inițiale, de dimensiuni mai mici, independente unele de altele (care folosesc mulțimi de date de intrare disjuncte – d_i).
- PAS2.** Dacă subproblema permite rezolvarea imediată (corespunde cazului de bază), **atunci** se rezolvă, obținându-se soluția s ; **altfel**, se revine la **Pas1**.
- PAS3.** Se **combină** soluțiile subproblemelor (s_i) în care a fost descompusă o subproblemă, **până când** se obține soluția problemei inițiale.

1.6.2. Implementarea metodei „Divide et Impera”

Deoarece subproblemele în care se descompune problema sunt similare cu problema inițială, algoritmul **divide et impera** poate fi implementat **recursiv**. Subprogramul recursiv **divide_et_impera(d,s)**, unde d reprezintă dimensiunea subproblemei (corespunde mulțimii datelor de intrare), iar s soluția subproblemei, poate fi descris în pseudocod astfel:

```

divide_et_impera(d,s)
început
    dacă dimensiunea  $d$  corespunde unui caz de bază
        atunci se determină soluția  $s$  a problemei;
    altfel
        pentru  $i=1,n$  execută
            se determină dimensiunea  $d_i$  a subproblemei  $P_i$ ;
            se determină soluția  $s_i$  a problemei  $P_i$  prin
                apelul divide_et_impera(d i,s i);
        sfârșit_pentru;

```

```

    se combină soluțiile s_1, s_2, s_3, ..., s_n;
    sfârșit_dacă;
sfârșit;

```

Implementarea acestui algoritm în limbajul C++ se face astfel:

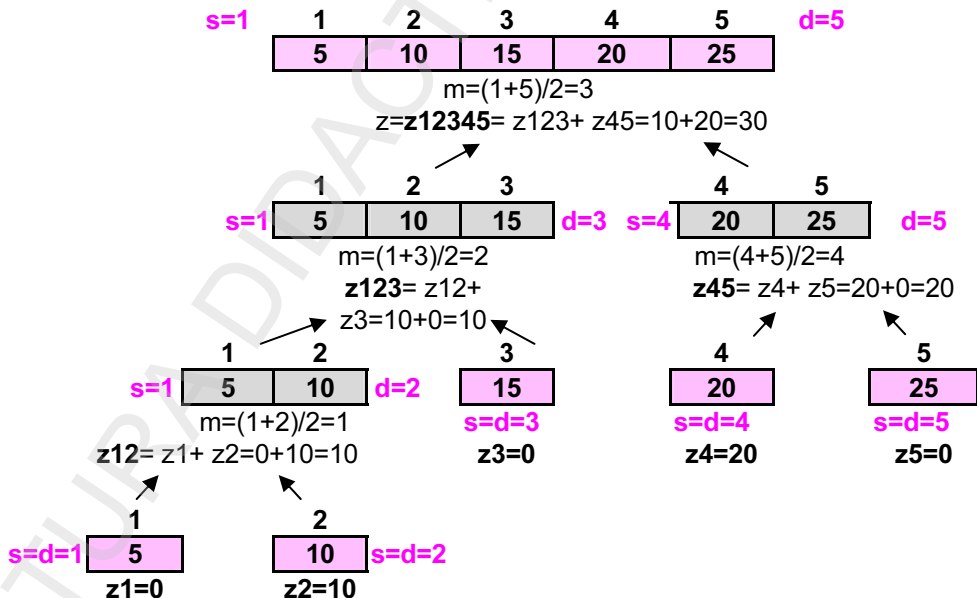
```

/*declarații globale pentru datele de intrare, ce vor fi divizate în sub-
mulțimi disjuncte pentru subproblemele în care se descompune problema*/
void divizeaza(<parametri: submulțimile>)
{//se divizează mulțimea de date de intrare în submulțimi disjuncte d_i}
void combina(<parametri: soluțiile s_i care se combină>)
{//se combină soluțiile obținute s_i}
void dei(<parametri: mulțimea de date d și soluția s>)
{//declarații de variabile locale
    if (<este caz de bază>) { //se obține soluția corespunzătoare subproblemei}
    else
    {divizeaza(<parametri: k submulțimi>);
      for (i=1;i=k;i++)
        dei(<parametri: mulțimea de date d i și soluția s i>);
      combina(<parametri: soluțiile s i>);}}

void main()
{//declarații de variabile locale
    //se citesc datele de intrare ale problemei - mulțimea d
    dei(<parametri: mulțimea de date d și soluția s>);
    //se afișează soluția problemei - s}

```

Exemplul 1. Să se calculeze **suma elementelor** pare **dintr-un vector v** care conține numere întregi. Numărul de elemente ale vectorului (**n**) și elementele lui se citesc de la tastatură.



Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$). Mulțimea datelor de intrare este divizată în două submulțimi disjuncte, prin divizarea mulțimii indicilor în două submulțimi disjuncte de indici, adică mulțimea indicilor **[s,d]** (unde **s** este primul indice, iar **d** ultimul indice) este divizată în două submulțimi disjuncte, **[s,m]** și **[m+1,d]**, unde **m** este indicele din mijlocul intervalului: $m=(s+d)/2$. În subprogram, procesul de divizare constă în determinarea mijlocului intervalului **m**.
- Subprogramul **combina()** – Combinarea soluției înseamnă adunarea celor două sume obținute prin rezolvarea celor două subprobleme. În subprogram sunt combinate cele două valori obținute din prelucrarea celor două intervale, adică se adună cele două valori **x** și **y**, obținându-se soluția **z**.
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submulțimea conține un singur element (se poate calcula suma, obținându-se soluția subproblemei). **Dacă** s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice) **atunci** se prelucrează cazul de bază (se calculează suma în variabila **z**, corespunzătoare soluției, astfel: **dacă** numărul **v[s]** este par, **atunci** suma va fi chiar numărul; **altfel**, are valoarea 0); **altfel**, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval, se apelează subprogramul **dei()** pentru al doilea interval și se combină cele două rezultate.

```
#include<iostream.h>
int v[100],n;
void divizeaza(int s,int d,int &m)
{m=(s+d)/2;}
void combina(int x,int y,int &z)
{z=x+y;}
void dei(int s,int d,int &z)
{int m,x1,x2;
 if (d==s)
  if (v[s]%2==0) z=v[s]; else z=0;
 else
  {divizeaza(s,d,m); dei(s,m,x1); dei(m+1,d,x2); combina(x1,x2,z);}}
void main()
{int i,z; cout<<"n= ";cin>>n;
 for(i=1;i<=n;i++) {cout<<"v["<<i<<"]=""; cin>>v[i];}
 dei(1,n,z); cout<<"suma= "<<z;}
```

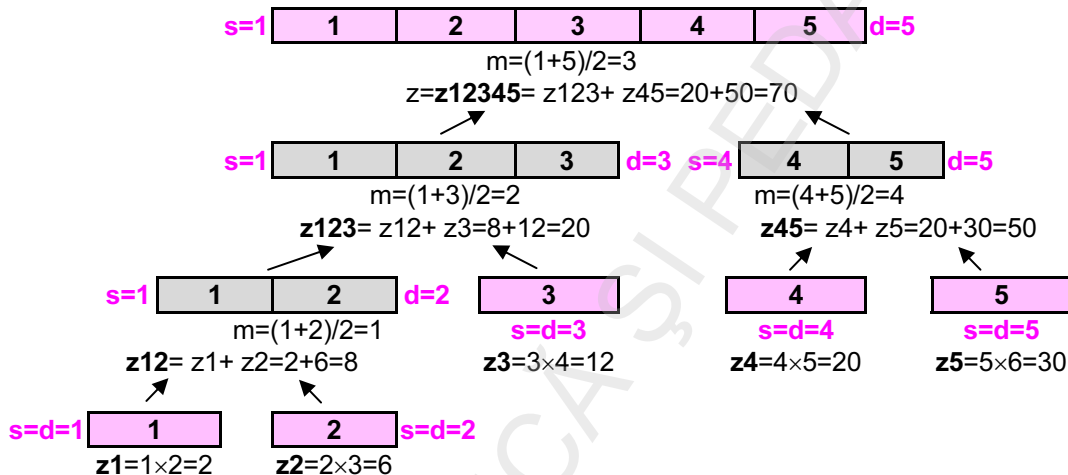
Exemplu 2: Să se calculeze suma $1 \times 2 + 2 \times 3 + \dots + n \times (n+1)$.

Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$). Mulțimea datelor de intrare este divizată în două submulțimi disjuncte, prin divizarea mulțimii primelor **n** numere naturale în două submulțimi disjuncte, adică mulțimea **[s,d]** (unde **s** este primul număr din mulțime, iar **d** ultimul număr din mulțime) este divizată în două submulțimi disjuncte, **[s,m]** și **[m+1,d]**, unde **m** este numărul din mijlocul intervalului: $m=(s+d)/2$. În subprogram, procesul de divizare constă în determinarea mijlocului intervalului, **m**.
- Subprogramul **combina()** – Combinarea soluției înseamnă adunarea celor două sume obținute prin rezolvarea celor două subprobleme. În subprogram sunt combinate cele

două valori obținute din cele două intervale (se adună cele două valori, x și y) obținându-se soluția z .

- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submulțimea conține un singur element (se poate calcula termenul sumei – produsul celor două numere consecutive – obținându-se soluția subproblemei). **Dacă** s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice), **atunci** se prelucerează cazul de bază (se calculează produsul, în variabila z , corespunzătoare soluției); **altfel**, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval, se apelează subprogramul **dei()** pentru al doilea interval și se combină cele două rezultate.



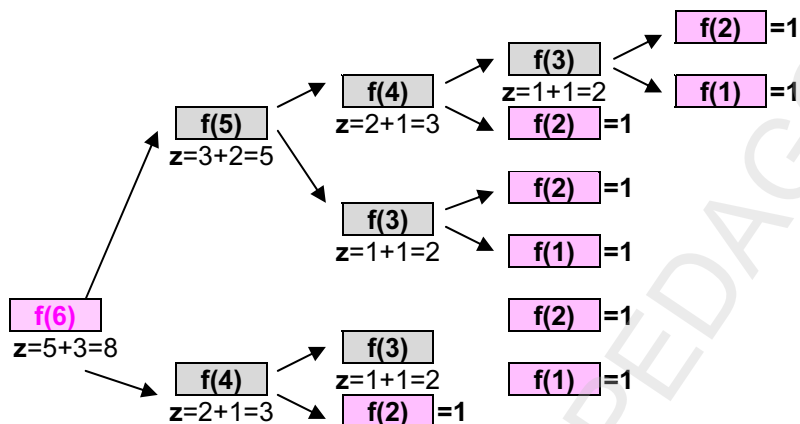
```
#include<iostream.h>
int n;
void divizeaza(int s,int d,int &m)
{m= (s+d) /2;}
void combina(int x,int y,int &z)
{z=x+y;}
void dei(int s,int d,int &z)
{int m,x1,x2;
 if (d==s) z=s*(s+1);
 else {divizeaza(s,d,m); dei(s,m,x1); dei(m+1,d,x2); combina(x1,x2,z);}}
void main()
{int z; cout<<"n= ";cin>>n; dei(1,n,z); cout<<"suma ="<<z; }
```

Exemplul 3: Să se calculeze termenul n al șirului lui Fibonacci.

Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 ($k=2$): determinarea termenului $n-1$ și determinarea termenului $n-2$. Descompunerea se face implicit, prin parametrul cu care se apelează subprogramul **dei()** – și subprogramul de divizare nu mai este necesar.
- Subprogramul **combina()** – Combinarea soluției înseamnă adunarea celor doi termeni ai șirului (x_1 și x_2) obținându-se soluția z .
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când s-a ajuns la un termen direct calculabil. **Dacă** s-a terminat procesul recursiv (prin procesul de divizare

s-a ajuns la termenul **f1** sau **f2**), **atunci** se prelucrează cazul de bază; **altfel**, se apelează subprogramul **dei()** pentru primul termen al descompunerii, se apelează subprogramul



dei() pentru al doilea termen al descompunerii și se combină cele două rezultate.

```

#include<iostream.h>
int n;
void combina(int x1,int x2,int &z)
{z=x1+x2;}
void dei(int n,int &z)
{int x1,x2;
 if (n==1 || n==2) z=1;
 else {dei(n-1,x1); dei(n-2,x2); combina(x1,x2,z);}}
void main()
{int z; cout<<"n= "; cin>>n; dei(n,z); cout<<z;}
  
```

Atenție

În acest exemplu, problema a fost descompusă în probleme care **nu sunt independente** unele de altele și, în apelurile recursive, aceeași subproblemă este rezolvată de mai multe ori: $f(4)$, $f(3)$, $f(2)$, $f(1)$.

Pentru acest tip de problemă **nu se recomandă** metoda divide et impera deoarece **nu este eficientă**. Pentru a calcula termenul f_n trebuie să se calculeze termenii f_{n-1} și f_{n-2} . Dar în calculul termenului f_{n-1} reapare calculul termenului f_{n-2} , iar în calculul ambilor termeni apare calculul termenului f_{n-3} . ș.a.m.d. În acest caz, se recomandă algoritmul iterativ, în care se calculează pe rând termenii f_3 , f_4 , ..., f_n , pornind de la termenii $f_1=f_2=1$. Formula de recurență este: $f_i = f_{i-1} + f_{i-2}$, pentru $3 \leq i \leq n$. Deoarece termenul f_i nu depinde decât de doi termeni, f_{i-1} și f_{i-2} , nu este necesar să se memoreze întregul șir de termeni, ci, pentru fiecare subproblemă i , numai termenii obținuți în subproblemele $i-1$ și $i-2$.

Exemplul 4: Să se determine, simultan, valoarea **minimă** și valoarea **maximă** dintr-un vector **v** care conține numere întregi. Numărul de elemente ale vectorului (**n**) și elementele lui se citesc de la tastatură.

Implementarea metodei **divide et impera** în acest exemplu se face astfel:

→ Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 (**k=2**). Divizarea mulțimii datelor de intrare se face la fel ca la exemplul pentru calcularea sumei elementelor unui vector – și subprogramele sunt identice.

- Subprogramul **combina()** – Combinarea soluției înseamnă determinarea minimului (**z1**) și a maximului (**z2**) dintre cele două valori minime (**x1** și **y1**), respectiv maxime (**x2** și **y2**) obținute prin rezolvarea celor două subprobleme. În subprogram sunt combinate cele două perechi de valori obținute din cele două intervale. **Dacă $x1 > y1$, atunci** minimul (**z1**) este **y1**; **altfel**, este **x1**. **Dacă $x2 > y2$, atunci** maximul (**z2**) este **x2**; **altfel**, este **y2**.
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submulțimea conține un singur element (se pot calcula minimul și maximul; atât minimul cât și maximul vor avea valoarea elementului). **Dacă** s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice), **atunci** se prelucrează cazul de bază (se calculează minimul și maximul, în variabilele **z1** și **z2** corespunzătoare soluției); **altfel**, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval, se apelează subprogramul **dei()** pentru al doilea interval – și se combină cele două rezultate.

```
#include<iostream.h>
int v[100],n;
void divizeaza(int s,int d,int &m)
{m=(s+d)/2;}
void combina(int x1,int y1,int &z1,int x2,int y2,int &z2)
{if (x1>y1) z1=y1; else z1=x1;
 if (x2>y2) z2=x2; else z2=y2;}
void dei(int s,int d,int &z1, int &z2) //z1 - minim, z2 - maxim
{int m,x1,x2,y1,y2;
 if (d==s) z1=z2=v[s];
 else {divizeaza(s,d,m);
       dei(s,m,x1,x2); //x1 - minim, x2 - maxim
       dei(m+1,d,y1,y2); //y1 - minim, y2 - maxim
       combina(x1,y1,z1,x2,y2,z2);}}
void main()
{int i,z1,z2; cout<<"n= ";cin>>n;
 for(i=1;i<=n;i++) {cout<<"v["<<i<<"]="; cin>>v[i];}
 dei(1,n,z1,z2);
 cout<<"minimul ="<<z1<<endl<<"maximul ="<<z2<<endl;}
```

Exemplul 5: Să se calculeze **suma a două polinoame**. Gradele celor două polinoame, **n** și **m**, și coeficienții celor două polinoame – se citesc de la tastatură. Coeficienții celor două polinoame se memorează în vectorii **p** și **q**, iar coeficienții polinomului sumă se memorează în vectorul **r**.

Implementarea metodei **divide et impera** în acest exemplu se face astfel:

- Subprogramul **divizeaza()** – Numărul de subprobleme în care se descompune problema este 2 (**k=2**). Deoarece există două mulțimi de date de intrare (vectorii **p** și **q**, care memorează coeficienții celor două polinoame), se va lua ca reper mulțimea cu cele mai multe elemente. Mulțimea datelor de intrare este divizată în două submulțimi disjuncte, prin divizarea mulțimii indicilor în două submulțimi disjuncte de indici, adică mulțimea indicilor **[s,d]** (unde **s** este primul indice, iar **d** este ultimul indice – **d=maxim(n,m)+1**) este divizată în două submulțimi disjuncte **[s,mijl]** și **[mijl+1,d]**, unde **mijl** este indicele din mijlocul intervalului: **mijl=(s+d)/2**. Procesul de divizare este identic cu cel de la exemplele anterioare.

- Subprogramul **combina()** – Deoarece în cazul de bază se determină unul dintre coeficienții polinomului sumă, care se scrie direct în vectorul **r**, acest subprogram nu mai este necesar.
- Subprogramul **dei()** – O subproblemă corespunde **cazului de bază** atunci când submulțimea conține un singur element (se poate calcula coeficientul polinomului sumă). **Dacă** s-a terminat procesul recursiv (prin procesul de divizare, cele două capete ale intervalului au ajuns să fie identice), **atunci** se prelucrează cazul de bază prin care se calculează coeficientul polinomului sumă pentru acel indice (**dacă** termenul care se calculează are indicele mai mic decât gradul minim al polinoamelor, **atunci** el este egal cu suma coeficienților celor două polinoame; **altfel, dacă** polinomul **p** are gradul mai mic decât al polinomului **q**, atunci el este egal cu coeficientul polinomului **q**; **altfel**, el este egal cu coeficientul polinomului **p**); **altfel**, se apelează subprogramul pentru divizarea intervalului, se apelează subprogramul **dei()** pentru primul interval – și apoi pentru al doilea interval.

```
#include<iostream.h>
int p[10],q[10],r[10],n,m;
int maxim(int x,int y)
{if (x>y) return x; else return y;}
int minim(int x,int y)
{if (x>y) return y; else return x;}
void divizeaza(int s,int d,int &mijl)
{mijl=(s+d)/2;}
void dei(int s,int d)
{int mijl;
 if (d==s)
   if (d<=minim(n,m)) r[d]=p[d]+q[d];
   else if (n<m) r[d]=q[d];
   else r[d]=p[d];
 else {divizeaza(s,d,mijl); dei(s,mijl); dei(mijl+1,d);}}
void main()
{int i; cout<<"n= ";cin>>n; cout<<"m= ";cin>>m;
 for(i=1;i<=n+1;i++) {cout<<"p("<<i-1<<"= "; cin>>p[i];}
 for(i=1;i<=m+1;i++) {cout<<"q("<<i-1<<"= "; cin>>q[i];}
 dei(1,maxim(n,m)+1);
 for(i=maxim(n,m)+1;i>=1;i--)
   if (r[i]!=0)
     {if (r[i]!=1) {cout<<r[i]; if (i!=1) cout<<"*";}
      if (i>2) cout<<"x^"<<i-1; else if (i==2) cout<<"x";
      if (i!=1) cout<<" + ";}}
```

Temă



1. Programul următor afișează – în ordine inversă – elementele unui vector. Explicați cum a fost folosită metoda **divide et impera** pentru a rezolva problema.

```
#include<iostream.h>
int v[100],n;
void divizeaza(int s,int d,int &m) {m=(s+d)/2;}
void dei(int s,int d)
{int m;
 if (d==s) cout<<v[s]<<"  ";
```



```

else {divizeaza(s,d,m); dei(m+1,d); dei(s,m);}}
void main()
{int i; cout<<"n= ";cin>>n;
for(i=1;i<=n;i++) {cout<<"a("<<i<<"= "; cin>>v[i];}
dei(1,n);}

```

2. Determinați ce calculează programul următor. Explicați cum a fost folosită metoda **divide et impera** pentru a rezolva problema.

```

#include<iostream.h>
int n;
void divizeaza(int s,int d,int &m) {m=(s+d)/2;}
void combina(int x,int y,int &z) {z=x+y;}
void dei(int s,int d,int &z)
{int m,x1,x2;
if (d==s) {if(s%2==0) z=-s*5; else z=s*5;}
else
{divizeaza(s,d,m); dei(s,m,x1); dei(m+1,d,x2); combina(x1,x2,z);}}
void main()
{int z; dei(1,20,z); cout<<"suma = "<<z;}

```

Temă



Scrieți următoarele programe, în care să folosiți metoda **divide et impera**. Valorile pentru datele de intrare se citesc de la tastatură.

1. Să se calculeze $n!$.
2. Să se calculeze – simultan – produsul și suma a n numere memorate într-un vector.
3. Să se calculeze suma $1+1 \times 2+1 \times 2 \times 3+...+1 \times 2 \times 3 \times ... \times n$.
4. Să se numere elementele pare dintr-un vector.
5. Să se verifice dacă un vector conține numai numere pozitive sau numai numere negative.
6. Să se calculeze c.m.m.d.c. a n numere memorate într-un vector.
7. Să se determine numărul de apariții ale unei valori x într-un vector.
8. Să se calculeze valoarea unui polinom $P(x)$, într-un punct x precizat.

Complexitatea algoritmului Divide et Impera

Metoda **divide et impera** se bazează pe rezolvarea recursivă a subproblemelor în care este divizată problema inițială.

Atunci când un algoritm conține un apel recursiv, timpul său de execuție este dat de o formulă recursivă care calculează timpul de execuție al algoritmului, pentru o dimensiune n a datelor de intrare, cu ajutorul timpilor de execuție pentru dimensiuni mai mici. Timpul de execuție al unui algoritm care folosește metoda divide et impera se bazează pe calculul timpilor de execuție ai celor trei etape de rezolvare a problemei. Dacă:

- problema inițială se divizează în a subprobleme, pentru care dimensiunea datelor de intrare reprezintă $1/b$ din dimensiunea problemei inițiale;
 - timpul de execuție al problemei inițiale este $T(n)$;
 - timpul necesar pentru a divide problema în subprobleme este $D(n)$;
 - timpul necesar pentru combinarea soluțiilor subproblemelor – pentru a obține soluția problemei – este $C(n)$;
 - timpul necesar pentru rezolvarea cazului de bază este $O(1)$;
- atunci se obține funcția pentru timpul de execuție – care este prezentată alăturat.

$$T(n) = \begin{cases} O(1) & \text{pentru cazul de bază} \\ a \times T(n/b) + D(n) + C(n) & \text{în caz contrar} \end{cases}$$

De exemplu, pentru a calcula suma elementelor unui vector cu n elemente, problema se descompune în două subprobleme ($a=2$) – și dimensiunea datelor de intrare pentru o subproblemă reprezintă jumătate din dimensiunea datelor inițiale ($b=2$). Pentru divizarea problemei în subprobleme, se calculează mijlocul intervalului de indici, și $O(D(n))=O(1)$. Pentru combinarea celor două soluții ale fiecărei subprobleme – adună cele două valori și $O(C(n))=O(1)$. Considerăm că $n=2^k$. Rezultă că:

$$\begin{aligned} T(n) &= T(2^k) + 2 \times O(1) = 2 \times (T(2^{k-1}) + 2 \times O(1)) + 2 \times O(1) = 2 \times (T(2^{k-1}) + 2^2 \times O(1)) = \\ &= 2 \times (2 \times (T(2^{k-2}) + 2 \times O(1)) + 2^2 \times O(1)) + 2^2 \times O(1) = 2^2 \times T(2^{k-2}) + 2^3 \times O(1) = \dots = \\ &= 2^k \times T(2^0) + 2^{k+1} \times O(1) = 2^k \times O(1) + 2^{k+1} \times O(1) = 2^k \times 3 = n \times 3. \end{aligned}$$

Ordinul de complexitate al algoritmului este $O(n)$. Algoritmul iterativ pentru rezolvarea acestei probleme are ordinul de complexitate $O(n)$.

Observație. Pentru acest gen de probleme, metoda iterativă este mai eficientă. Rezolvarea acestui tip de probleme cu ajutorul metodei divide et impera a avut numai un rol scolastic, pentru înțelegerea metodei și a implementării ei.

Metoda divide et impera **se recomandă** în următoarele cazuri:

- algoritmul obținut este **mai eficient** decât algoritmul clasic (iterativ) – de exemplu, algoritmul de căutare într-un vector sortat și algoritmi pentru sortarea unui vector;
- rezolvarea problemei prin divizarea ei în subprobleme este **mai simplă** decât rezolvarea clasică (iterativă) – de exemplu, problema turnurilor din Hanoi.

1.6.3. Căutarea binară

Să se caute – într-un șir de numere întregi ordonate strict crescător (sau varianta strict descrescător) – poziția în care se găsește, în șir, o valoare x citită de la tastatură. Dacă valoarea nu se găsește în șir, să se afișeze un mesaj de informare.

Algoritmul de căutare secvențială într-un vector are ordinul de complexitate $O(n)$. Pornind de la faptul că vectorul este un vector particular (este ordonat strict crescător sau strict descrescător) se poate folosi metoda **divide et impera**. Pentru un vector ordonat strict crescător, **pașii algoritmului sunt**:

PAS1. Se **divizează** vectorul v în doi subvectori, prin divizarea mulțimii indicilor $[s,d]$ (unde s este indicele primului element, iar d indicele ultimului element), în două submulțimi disjuncte, $[s,m-1]$ și $[m+1,d]$, unde m este indicele din mijlocul intervalului: $m=(s+d)/2$.

PAS2. Dacă elementul situat pe poziția din mijloc ($v[m]$) are valoarea x , atunci problema este rezolvată – și poziția este m ; **altfel, dacă** elementul din mijlocul vectorului este mai mic decât valoarea x , atunci căutarea se face printre elementele cu indicii în mulțimea $[s,m]$; **altfel**, căutarea se face printre elementele cu indicii din mulțimea $[m+1,d]$. Pasul 2 se execută până când se găsește elementul sau până când vectorul nu mai poate fi împărțit în subvectori.

```
#include<iostream.h>
int v[100],n,x;
void divizeaza(int s,int d,int &m)
{m=(s+d)/2;}
void caută(int s,int d,int &z)
{int m;
 if (d>s){divizeaza(s,d,m);
  if (v[m]==x) z=m;
```

```

        else if (x>v[m]) cauta(m+1,d,z);
                else cauta(s,m-1,z);}}

void main()
{int i,z=0; cout<<"n= ";cin>>n; cout<<"x= ";cin>>x;
  for(i=1;i<=n;i++) {cout<<"v["<<i<<"]="; cin>>v[i];}
  cauta(1,n,z);
  if(z==0) cout<<"nu exista elementul "<<x<<" in vector";
    else cout<<"exista pe pozitia "<<z;}
```

Complexitatea algoritmului de căutare binară. Pentru divizarea problemei în subprobleme, se calculează mijlocul intervalului de indici și $O(D(n))=O(1)$. Deoarece căutarea se face numai într-unul dintre cei doi subvectori (problema inițială se rezolvă prin rezolvarea uneia dintre cele două subprobleme) și $a=1$, formula recurentă a timpului de execuție este $T(n)=T(n/2)+O(1)$. Considerăm că $n=2^k$ ($k=\log_2 n$). Rezultă că:

$$T(n)=T(2^k)+O(1) = (T(2^{k-1})+O(1))+O(1)=T(2^{k-1})+2\times O(1) = \dots = T(2^0)+(k+1)\times O(1) = O(1)+(k+1)\times O(1) = (k+2)\times O(1) = \log_2 n \times O(1).$$

Ordinul de complexitate al algoritmului este $O(\log_2 n)$.

Aplicarea algoritmului de căutare binară

Exemplu. Să se determine – cu o precizie de 4 zecimale – rădăcina reală din intervalul $[0,1]$ a ecuației $x^3+x-1=0$.

S-a identificat pentru această ecuație o rădăcină în intervalul $[0,1]$ și ne propunem să localizăm această rădăcină în limitele unei precizii de 4 zecimale, printr-o valoare x . Pentru căutarea valorii x , se va folosi **metoda biseției** – care constă în reducerea intervalului de căutare prin înjumătățirea repetată și selectarea subintervalului în care se găsește rădăcina. Intervalul $[s,d]$ este împărțit în două subintervale, $[s,m]$ și $[m,d]$, unde $m=(s+d)/2$. Căutarea rădăcinii se va face în subintervalul în care funcția $f(x)=x^3+x-1$ își schimbă semnul, astfel: **dacă** $f(s)*f(m)<0$, **atunci** căutarea continuă în intervalul $[s,m]$; **altfel**, căutarea continuă în subintervalul $[m,d]$. Procesul recursiv este întrerupt când se ajunge la intervalul $[s,d]$ pentru care $d-s<r$, unde r este eroarea acceptată pentru o precizie de 4 zecimale și are valoarea 0,0001.

```

#include<iostream.h>
#include<iosmanip.h>
const float r=0.0001;
float f(float x)
{return x*x*x+x-1;}
void divizeaza(float s,float d,float &m)
{m=(s+d)/2;}
void radacina(float s,float d,float &z)
{float m;
  if (d-s<r) z=(s+d)/2;
  else
  {divizeaza(s,d,m);
   if (f(s)*f(m)<0) radacina(s,m,z);
   else radacina(m,d,z);}}
```

```

void main()
{float z=0; radacina(0,1,z); cout<<"radacina= "<<z<<endl;
  cout<<"f(x)= "<<setiosflags(ios::fixed)<<f(z); }
```

Temă



1. Să se caute – într-un șir de numere întregi, în care mai întâi se găsesc numerele pare și apoi numerele impare – poziția în care se găsește, în șir, o valoare x citită de la tastatură.
2. Să se calculeze radicalul de ordinul 2 din numărul a , cu o aproximație de 4 zecimale, fără a folosi funcția matematică de sistem `sqrt()`. Să se compare rezultatul obținut cu rezultatul furnizat de funcția matematică de sistem. **Indicație.** Pornind de la ecuația $x^2 - a = 0$ se va identifica intervalul în care se găsește soluția și apoi se va localiza cu o precizie de 4 zecimale rădăcina reală din acest interval.
3. Să se calculeze partea întreagă a radicalului de ordinul 3 din numărul a , fără a folosi funcția matematică de sistem.

1.6.4. Sortarea prin interclasare (MergeSort)

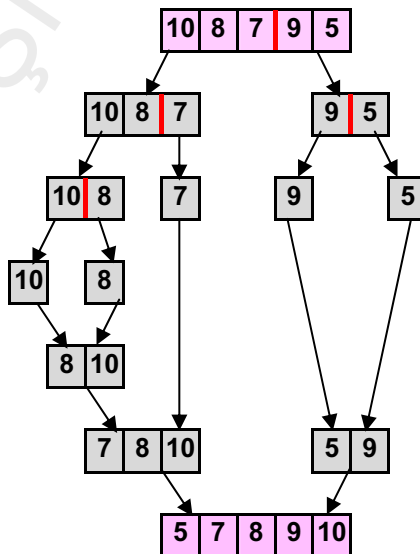
Algoritmul de interclasare se execută pe doi vectori ordonați după același criteriu, pentru a obține un al treilea vector, care să conțină elementele primilor doi vectori, ordonate după același criteriu. Algoritmul de sortare prin interclasare se bazează pe observația că orice vector care conține un singur element este un vector sortat. Algoritmul de interclasare se poate folosi pentru sortarea unui vector, folosind metoda **divide et impera**, astfel:

PAS1. Se **descompune** problema în subprobleme similare, prin împărțirea vectorului în doi subvectori având mulțimea indicilor $[s,m]$ și $[m+1,d]$, unde m este indicele din mijlocul intervalului: $m=(s+d)/2$.

PAS2. Dacă subvectorul conține un singur element, **atunci** se consideră sortat (corespunde cazului de bază); **altfel**, se continuă descompunerea lui în subvectorii care au mulțimea indicilor $[s,m]$ și mulțimea indicilor $[m+1,d]$.

PAS3. Se **combină** soluțiile celor două subprobleme prin **interclasarea** celor doi vectori sortați, obținându-se un vector sortat.

Vectorul care se sortează este x . Prin subprogramul **interclaseaza()** se realizează combinarea soluțiilor, prin interclasarea subvectorului x , care are mulțimea indicilor $[s,m]$, cu subvectorul x , care are mulțimea indicilor $[m+1,d]$, în vectorul auxiliar v , după care vectorul v se copiază în vectorul x care are mulțimea indicilor $[s,d]$.



```
#include<iostream.h>
int x[100],n;
void divizeaza(int s,int d,int &m)
{m=(s+d)/2;}
void interclaseaza(int s,int d,int m)
{int i=s,j=m+1,k=1,v[100];
 while (i<=m && j<=d)
 {if (x[i]<x[j]) {v[k]=x[i]; i++;}
  else {v[k]=x[j]; j++;}
  k++;}
 if (i<=m) while (i<=m) {v[k]=x[i]; i++; k++;}
 if (j<=d) while (j<=d) {v[k]=x[j]; j++; k++;}
```

```

    else while (j<=d) {v[k]=x[j]; j++; k++;}
    for (k=1,i=s;i<=d;k++,i++) x[i]=v[k];}

void MergeSort(int s,int d)
{int m;
 if (s<d) {divizeaza(s,d,m);
           MergeSort(s,m);
           MergeSort(m+1,d);
           interclaseaza(s,d,m);}}

void main()
{int i; cout<<"n= ";cin>>n;
 for(i=1;i<=n;i++) {cout<<"x["<<i<<"]= ";cin>>x[i];}
 MergeSort(1,n);
 cout<<"vectorul sortat"<<endl;
 for(i=1;i<=n;i++) cout<<x[i]<<" ";}

```

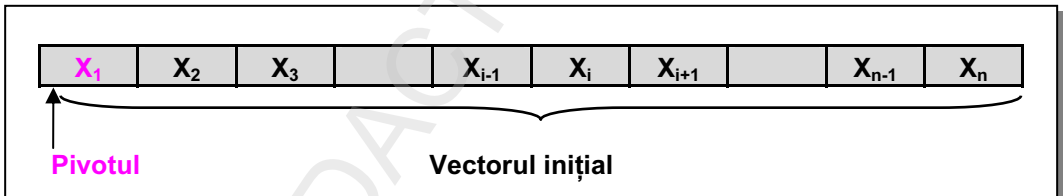
Complexitatea algoritmului de sortare prin interclasare. Pentru divizarea problemei în subprobleme, se calculează mijlocul intervalului de indici și $O(D(n))=O(1)$. Pentru combinarea soluțiilor, se execută interclasarea a doi vectori și $O(D(n))=O(n)$. Timpul de execuție este $T(n)=2 \times T(n/2)+n$. Considerând că $n=2^k$, rezultă:

$$T(n)=T(2^k)+2^k=2 \times (T(2^{k-1})+2^{k-1})+2^k=2 \times T(2^{k-1})+2^k+2^k=\dots=2 \times 2 \times (T(2^{k-2})+2^{k-2})+2^k+2^k=2^2 \times (T(2^{k-2})+2^{k-2})+2^k+2^k+2^k=k \times 2^k=\log_2 n \times n.$$

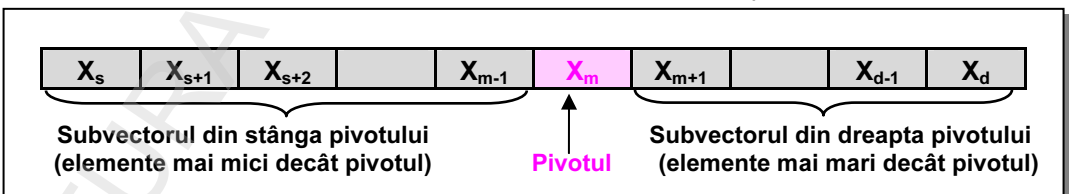
Ordinul de complexitate al algoritmului este $O(n \times \log_2 n)$.

1.6.5. Sortarea rapidă (QuickSort)

Prin această metodă de sortare se execută următoarele operații prin care sunt rearanjate elementele din cadrul vectorului:



- Primul element din vector, numit **pivot**, este mutat în cadrul vectorului pe poziția pe care trebuie să se găsească în vectorul sortat.
- Toate elementele mai mici decât el vor fi mutate în vector în fața sa.
- Toate elementele mai mari decât el vor fi mutate în vector după el.



De exemplu, dacă vectorul conține elementele **{3, 4, 1, 5, 2}**, după executarea operațiilor precizate vectorul va fi **{2, 1, 3, 5, 4}**.

Folosind metoda **divide et impera** problema inițială va fi descompusă în subprobleme astfel:

PAS1. Se rearanjează vectorul, determinându-se poziția în care va fi mutat pivotul (**m**).

PAS2. Problema inițială (sortarea vectorului inițial) se descompune în două subprobleme, prin descompunerea vectorului în doi subvectori: vectorul din stânga pivotului și vectorul din dreapta pivotului – care vor fi sortați prin aceeași metodă. Acești subvectori, la rândul lor, vor fi și ei rearanjați și împărțiți de pivot în doi subvectori etc.

PAS3. Procesul de descompunere în subprobleme va continua până când, prin descompunerea vectorului în subvectori, se vor obține vectori care conțin un singur element.

Subprogramele specifice algoritmului **Divide et Impera** vor avea următoarea semnificație:

→ În subprogramul **divizeaza()** se va rearanja vectorul și se va determina poziția pivotului x_m , care va fi folosită pentru divizarea vectorului în doi subvectori: $[x_s, x_{m-1}]$ și $[x_{m+1}, x_d]$.

→ Subprogramul **combina()** nu mai este necesar, deoarece combinarea soluțiilor se face prin rearanjarea elementelor în vector.

În subprogramul **divizeaza()** vectorul se parcurge de la ambele capete către poziția în care trebuie mutat pivotul. Se vor folosi doi indici: **i** – pentru parcurgerea vectorului de la începutul lui către poziția pivotului (**i** se va incrementa) și **j** – pentru parcurgerea vectorului de la sfârșitul lui către poziția pivotului (**j** se va decrementa). Cei doi indici vor fi inițializați cu capetele vectorului (**i=s**, respectiv **j=d**) și se vor deplasa până când se întâlnesc, adică atât timp cât **i<j**. În momentul în care cei doi indici s-au întâlnit înseamnă că operațiile de rearanjare a vectorului s-au terminat – și pivotul a fost adus în poziția corespunzătoare lui în vectorul sortat. Această poziție este **i** (sau **j**) și va fi poziția **m** de divizare a vectorului.

În exemplele următoare sunt prezentate două versiuni pentru subprogramul **divizeaza()**:

Versiunea 1. Se folosesc variabilele logice: **pi**, pentru parcurgerea cu indicele **i**, și **pj**, pentru parcurgerea cu indicele **j**: Ele au valoarea: 1 – se parcurge vectorul cu acel indice și 0 – nu se parcurge vectorul cu acel indice; cele două valori sunt complementare.

```
#include<iostream.h>
int x[100],n;
void schimb(int &a, int &b)
{int aux=a; a=b; b=aux;}
void divizeaza(int s,int d,int &m)
{int i=s,j=d,pi=0,pj=1;
// pivotul fiind pe pozitia s, parcurgerea incepe cu indicele j
while (i<j)
    {if (x[i]>x[j]) {schimb(v[i],v[j]); schimb(pi,pj);}
    i=i+pi; j=j-pj;}
m=i;}
void QuickSort(int s,int d)
{int m;
if (s<d) {divizeaza(s,d,m);
QuickSort(s,m-1);
QuickSort(m+1,d);}}
void main()
{int i; cout<<"n= ";cin>>n;
for(i=1;i<=n;i++) {cout<<"x["<<i<<"]= ";cin>>x[i];}
QuickSort(1,n);
cout<<"vectorul sortat"<<endl;
for(i=1;i<=n;i++) cout<<x[i]<<" ";
```

Cei doi indici – i și j – sunt inițializați cu extremitățile vectorului ($i=1$; $j=5$) și parcurgerea începe cu indicele j ($pi=0$; $pj=1$)

Se compară pivotul (3) cu ultimul element (2). Deoarece pivotul este mai mic, cele două valori se interschimbă – și se schimbă și modul de parcurgere ($pi=1$; $pj=0$ – avansează indicele i).

Se compară elementul din poziția i (4) cu elementul din poziția j (3). Deoarece 4 este mai mare decât 3, cele două valori se interschimbă – și se schimbă și modul de parcurgere ($pi=0$; $pj=1$ – avansează indicele j).

Se compară elementul din poziția i (3) cu elementul din poziția j (5). Deoarece 3 este mai mic decât 5, cele două valori nu se interschimbă și se păstrează modul de parcurgere ($pi=0$; $pj=1$ – avansează indicele j).

Se compară elementul din poziția i (3) cu elementul din poziția j (1). 3 fiind mai mare decât 1, cele două valori se interschimbă și se schimbă și modul de parcurgere (avansează indicele i). Cei doi indici sunt egali.

i				j
1	2	3	4	5
3	4	1	5	2

	i			j
1	2	3	4	5
2	4	1	5	3

	i		j	
1	2	3	4	5
2	3	1	5	4

	i		j	
1	2	3	4	5
2	3	1	5	4

		i j		
1	2	3	4	5
2	1	3	5	4

Versiunea 2:

```
void divizeaza(int s,int d,int &m)
{int pivot=v[s],i=s,j=d;
  while (i<j) {while(v[i]<pivot) i++;
               while(v[j]>pivot) j--;
               if (i<j) schimb(v[i],v[j]);}
  m=i;}
```

Inițial, cei doi indici – i și j – sunt inițializați cu extremitățile vectorului ($i=1$; $j=5$) și pivotul are valoarea 3.

Elementul din poziția i (3) nu este mai mic decât pivotul; indicele i nu avansează ($i=1$). Elementul din poziția j (2) nu este mai mare decât pivotul; indicele j nu avansează ($j=5$). Valorile din pozițiile i și j se interschimbă.

Elementul din poziția i (2) este mai mic decât pivotul; indicele i avansează până la primul element mai mare decât pivotul ($i=2$). Elementul din poziția j (3) nu este mai mare decât pivotul; indicele j ($j=5$) nu avansează. Valorile din pozițiile i și j se interschimbă.

Elementul din poziția i (3) nu este mai mic decât pivotul; indicele i nu avansează ($i=2$). Elementul din poziția j (4) este mai mare decât pivotul; indicele j avansează până la primul element mai mic decât pivotul ($j=3$). Valorile din pozițiile i și j se interschimbă.

Elementul din poziția i (1) este mai mic decât pivotul; indicele i avansează la primul element mai mare decât pivotul ($i=4$). Elementul din poziția j (3) nu este mai mare decât pivotul; indicele j nu avansează ($j=3$). Indicele i este mai mare decât indicele j .

i				j
1	2	3	4	5
3	4	1	5	2

i				j
1	2	3	4	5
2	4	1	5	3

	i		j	
1	2	3	4	5
2	3	1	5	4

	i		j	
1	2	3	4	5
2	1	3	5	4

		j	i	
1	2	3	4	5
2	1	3	5	4

Observație. În ambele cazuri, algoritmul continuă cu divizarea vectorului în subvectorii cu indicii [1,2] și [4,5] și rearanjarea elementelor în cei doi subvectori.

Complexitatea algoritmului de sortare rapidă. Pentru divizarea problemei în subprobleme se calculează mijlocul intervalului de indici și $O(D(n))=O(1)$. Pentru combinarea soluțiilor se parcurge vectorul, cu ajutorul celor doi indici, de la primul element până la ultimul element, și $O(D(n))=O(n \times O(1))=O(n)$. Timpul de execuție este $T(n)=2 \times T(n/2)+n$. Considerând că $n=2^k$, rezultă:

$$T(n)=T(2^k)+2^k=2 \times (T(2^{k-1})+2^{k-1})+2^k=2 \times T(2^{k-1})+2^k+2^k=\dots=2 \times 2 \times (T(2^{k-2})+2^{k-2})+2^k+2^k=2^2 \times (T(2^{k-2})+2^{k-2})+2^k+2^k+2^k=k \times 2^k=\log_2 n \times n.$$

Ordinul de complexitate al algoritmului este $O(n \times \log_2 n)$.

Observație. În comparație cu algoritmi de sortare prin metoda selecției directe și prin metoda bulelor, care au ordinul de complexitate $O(n^2)$, algoritmi de sortare care folosesc strategia **divide et impera** sunt mai eficienți.

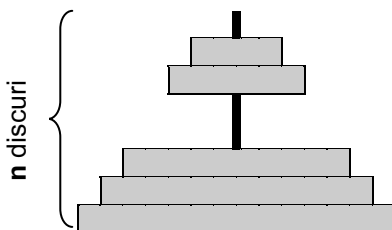


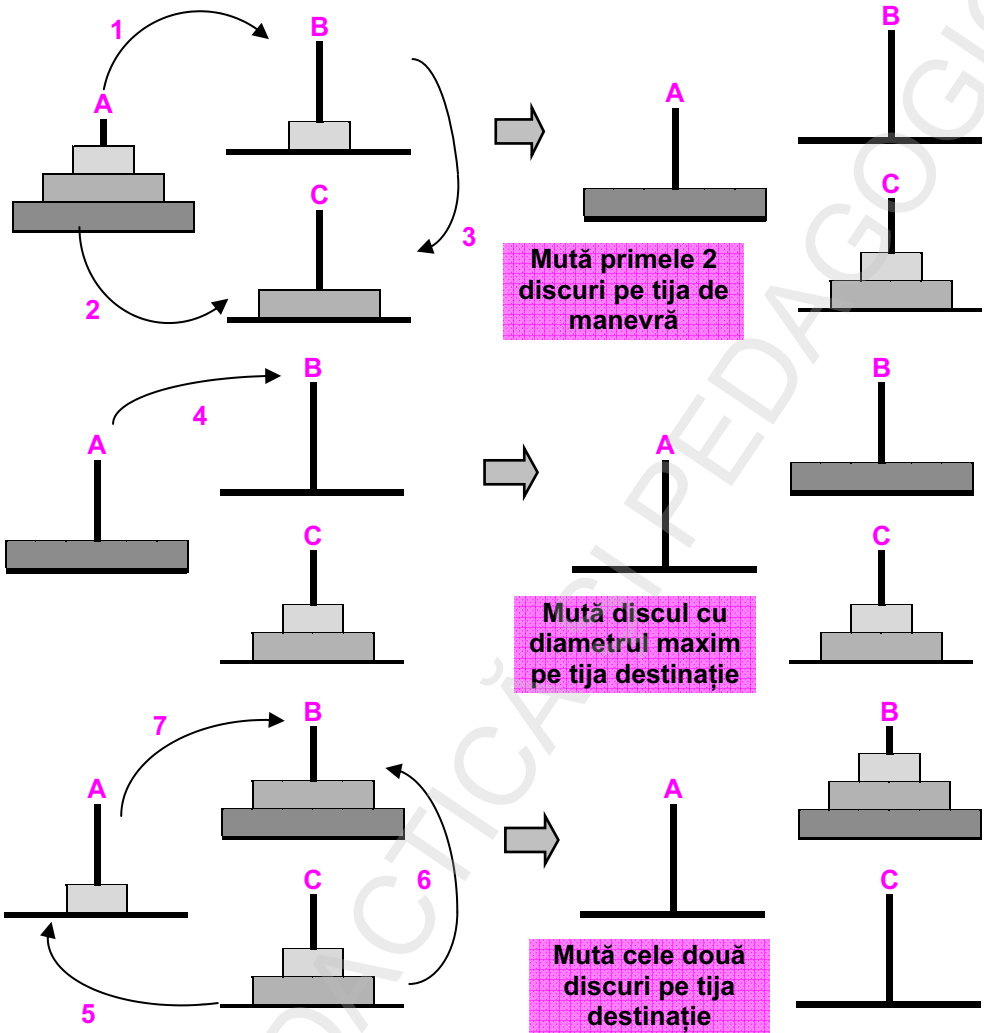
1. Să se rearanjeze elementele unui vector astfel încât în vector să se găsească mai întâi numerele impare și apoi numerele pare. Se vor realiza două versiuni ale programului, câte una pentru fiecare metodă de sortare care folosește **strategia divide et impera**.
2. Într-un fișier text sunt scrise următoarele informații: pe primul rând numărul de elevi din clasă – n , iar pe următoarele n rânduri, pentru fiecare elev, mediile semestriale la disciplina informatică. În cadrul unui rând, mediile sunt separate prin spațiu. Fiecare elev se identifică prin numărul de ordine la citirea din fișier. Să se scrie un program care să realizeze următoarele cerințe. Fiecare cerință va fi rezolvată printr-un subprogram. Ordonarea elementelor vectorului și căutarea unui elev se vor face folosind algoritmi cei mai eficienți.
 - a. Se citesc datele din fișierul text și se calculează media anuală a fiecărui elev.
 - b. Pentru identificatorul unui elev, citit de la tastatură, se afișează mediile semestriale și media anuală.
 - c. Se rearanjează elevii în ordinea crescătoare a mediilor și se afișează mediile fiecărui elev.
 - d. Se scriu informațiile obținute în urma prelucrărilor într-un alt fișier text.

1.6.6. Problema turnurilor din Hanoi

Pe trei tije, notate cu **A**, **B** și **C**, se pot monta discuri perforate, de diferite dimensiuni. Inițial, pe tija **A** (tija sursă) sunt așezate unele peste altele n discuri, în ordinea descrescătoare a diametrelor, iar celelalte tije sunt goale. Să se afișeze toate mutările care trebuie făcute ca discurile de pe tija **A** să fie aranjate pe tija **B** (tija destinație) în aceeași ordine în care erau pe tija **A**, folosind tija **C** ca tijă de manevră. O mutare nu se poate face decât cu un disc – și el nu poate fi așezat pe o tijă decât peste un disc cu diametru mai mare. Să analizăm mutările care trebuie făcute pentru $n=3$:

- Se mută discul de pe tija **A** pe tija **B** (1).
- Se mută discul de pe tija **A** pe tija **C** (2).
- Se mută discul de pe tija **B** pe tija **C** (3).
- Se mută discul de pe tija **A** pe tija **B** (4).
- Se mută discul de pe tija **C** pe tija **A** (5).
- Se mută discul de pe tija **C** pe tija **B** (6).
- Se mută discul de pe tija **A** pe tija **B** (7).





Folosind metoda **Divide et Impera** problema inițială va fi descompusă în subprobleme astfel:

PAS1. Se mută primele $n-1$ discuri de pe tija sursă pe tija de manevră.

PAS2. Se mută discul cu diametrul cel mai mare de pe tija sursă pe tija destinație.

PAS3. Se mută cele $n-1$ discuri de pe tija de manevră pe tija destinație.

```
#include<iostream.h>
void hanoi(int n, char a, char b, char c)
{if (n==1) cout<<"Mutarea: "<<a<<"->"<<b<<endl;
  else
  {hanoi(n-1,a,c,b);
   cout<<"Mutarea: "<<a<<"->"<<b<<endl;
   hanoi(n-1,c,b,a);}}
void main()
{int n; char a='A',b='B',c='C'; cout<<"n= "; cin>>n;
  hanoi(n,a,b,c);}
```



Într-un parc cu o suprafață dreptunghiulară care are coordonatele colțului din stânga sus (X_1, Y_1) și ale colțului din dreapta jos (X_2, Y_2) se găsesc n copaci. Poziția fiecărui copac pe teren este dată de coordonatele (x_i, y_i) . În parc, trebuie amenajat un teren de joacă pentru copii, de formă dreptunghiulară, cu laturile paralele cu laturile parcului. Să se determine dreptunghiul cu arie maximă pe care se poate amenaja terenul de joacă fără să se taie nici un copac. **Indicație.** Considerând că un copac i împarte dreptunghiul inițial în patru dreptunghiuri cu laturile paralele cu dreptunghiul inițial, problema se descompune în patru subprobleme, fiecare subproblemă rezolvând cazul unuia dintre cele patru dreptunghiuri, care au coordonatele colțurilor $((X_1, Y_1); (x_i, Y_2))$, $((x_i, Y_1); (X_2, Y_2))$, $((X_1, Y_1); (X_2, y_i))$ și $((X_1, y_i); (X_2, Y_2))$. La cazul de bază se ajunge atunci când dreptunghiul nu conține nici un copac. Pentru fiecare dreptunghi care nu conține nici un copac, se calculează aria și se memorează coordonatele colțurilor. Din combinarea soluțiilor obținute – se identifică dreptunghiul care are aria cea mai mare.

Evaluate

Adevărat sau Fals:

1. În algoritmul metodei backtracking, inițializarea elementului k al soluției se face când se revine de pe nivelul $k+1$ la nivelul k .
2. Algoritmul metodei backtracking se încheie dacă s-au testat toate valorile posibile pentru primul element al soluției.
3. În algoritmul backtracking, după găsirea unei soluții, se revine la nivelul anterior al soluției.
4. În algoritmul backtracking, trecerea de la nivelul k la nivelul $k-1$ al soluției se face după ce s-au găsit toate valorile posibile pentru nivelul k .
5. În algoritmul backtracking, inițializarea nivelului k al soluției se face cu valoarea de pe nivelul anterior al soluției.
6. În algoritmul backtracking, dacă s-a trecut la nivelul k al soluției înseamnă că s-au găsit primele $k-1$ elemente ale soluției.
7. În algoritmul backtracking, trecerea la nivelul $k+1$ al soluției se face după ce au fost testate toate valorile posibile pentru nivelul k al soluției.
8. În implementarea recursivă a metodei backtracking, revenirea din autoapel este echivalentă cu revenirea la nivelul anterior al soluției, din varianta iterativă.
9. În implementarea recursivă a metodei backtracking, autoapelul este echivalent cu trecerea la nivelul următor al soluției din varianta iterativă.
10. Divide et impera este metoda prin care problema inițială se descompune în subprobleme care nu sunt independente.
11. Prin metoda divide et impera se găsește soluția optimă a problemei.

Alegeți:

1. Algoritmul care implementează metoda backtracking este:
a. polinomial b. logaritmic c. exponențial d. liniar logaritmic
2. Algoritmul care implementează metoda sortării rapide este:
a. liniar b. logaritmic c. pătratic d. liniar logaritmic
3. Algoritmul care implementează metoda căutării binare este:
a. liniar b. logaritmic c. pătratic d. liniar logaritmic

4. Se utilizează metoda backtracking pentru generarea anagramelor cuvântului *masca*. Să se precizeze cuvântul precedent și cuvântul următor secvenței *camas, camsa, caams*:
- a. csaam și casma b. csama și casma c. csaam și casma d. csaam și caasm
5. Se utilizează metoda backtracking pentru generarea tuturor numerelor care se pot forma cu cifrele 0, 2, 4 și 8. Să se precizeze numărul precedent și numărul următor secvenței 2840 4028 4082:
- a. 2480 și 4280 b. 2804 și 4280 c. 2804 și 4208 d. 2480 și 4280
6. Se utilizează metoda backtracking pentru generarea șirului de opt paranteze rotunde, astfel încât ele să formeze o succesiune corectă. Să se precizeze șirul de paranteze precedent și șirul de paranteze următor secvenței $((()()) (())()) (())()$:
- a. $((()()))$ și $()((()))$ b. $((()))()$ și $()((()))$ c. $((()))()$ și $((())())$ d. $((()()))$ și $((())())$
7. Se utilizează metoda backtracking pentru generarea submulțimilor mulțimii $\{1,2,3\}$. Să se precizeze submulțimea precedentă și submulțimea următoare secvenței de submulțimi $\{1\} \{1,3\} \{1,2\}$:
- a. $\{\}$ și $\{2,3\}$ b. $\{\}$ și $\{2\}$ c. $\{2,3\}$ și $\{1,2,3\}$ d. $\{\}$ și $\{3\}$
8. Dacă se utilizează metoda backtracking pentru a genera toate permutările de 5 obiecte și primele 4 permutări generate sunt: 5 4 3 2 1, 5 4 3 1 2, 5 4 2 3 1, 5 4 2 1 3, atunci a cincea permutare este:
- a. 5 4 3 2 1 b. 5 3 4 2 1 c. 5 4 1 2 3 d. 5 4 1 3 2
- (Bacalaureat – Sesiunea specială 2003)
9. Se cere determinarea tuturor modalităților de planificare în zile diferite a 4 probe (rezistență, aruncarea suliței, sărituri, tir) în oricare dintre cele 7 zile din săptămână. Problema este echivalentă cu:
- a. generarea combinațiilor de 4 obiecte luate câte 7
b. generarea aranjamentelor de 4 obiecte luate câte 7
c. generarea combinațiilor de 7 obiecte luate câte 4
d. generarea aranjamentelor de 7 obiecte luate câte 4
- (Bacalaureat – Sesiunea specială 2003)
10. Generarea tuturor șirurilor de 3 elemente, fiecare element putând fi oricare număr din mulțimea $\{1,2,3,4,5,6\}$, se realizează cu ajutorul unui algoritm echivalent cu algoritmul de generare a:
- a. aranjamentelor b. permutărilor c. combinațiilor d. produsului cartezian
- (Bacalaureat – Sesiunea specială 2003)
11. Se cere determinarea tuturor modalităților distincte de așezare în linie a tuturor celor n sportivi aflați la o festivitate de premiere. Problema este echivalentă cu generarea:
- a. partițiilor unei mulțimi de n obiecte b. aranjamentelor de n obiecte luate câte 1
c. permutărilor de n obiecte d. submulțimilor unei mulțimi cu n obiecte
- (Bacalaureat – Sesiunea iunie-iulie 2003)
12. Se cere determinarea tuturor numerelor formate cu cifre aflate în ordine strict crescătoare, cifre alese dintr-o mulțime cu k cifre distincte date. De exemplu, pentru cifrele alese din mulțimea $\{1,5,2\}$ se obțin numerele 1, 2, 12, 5, 15, 25, 125 (nu neapărat în această ordine). Problema este echivalentă cu generarea:
- a. partițiilor unei mulțimi b. aranjamentelor de 10 obiecte luate câte k
c. permutărilor de k obiecte d. submulțimilor nevide ale unei mulțimi
- (Bacalaureat – Sesiunea august 2003)

13. Cineva dorește să obțină și să prelucreze toate numerele formate din trei cifre din șirul 2, 9, 4 și le generează exact în ordinea 222, 229, 224, 292, 299, 294, 242, 249, 244, 922, 929, 924, 992, 999, 994, 942, 949, 944, 422, 429, 424, 492, 499, 494, 442, 449, 444. Dacă dorește să obțină prin același procedeu numerele formate din 4 cifre, atunci după numărul 4944 va urma:
- a. 4949 b. 9222 c. 4422 d. 4924
(Bacalaureat – Simulare 2004)
14. Un elev folosește metoda backtracking pentru a genera submulțimile mulțimii {1,2,5,6,9}. Câte soluții (submulțimi) care obligatoriu conțin elementul 2 și nu conțin elementul 6 a generat?
- a. 7 b. 16 c. 6 d. 8
(Bacalaureat – Sesiunea specială 2004)
15. Pentru a determina toate modalitățile de a scrie pe 9 ca sumă de numere naturale nenule distincte (abstracție făcând de ordinea termenilor), un elev folosește metoda backtracking generând, în această ordine, toate soluțiile: 1+2+6, 1+3+5, 1+8, 2+3+4, 2+7, 3+6 și 4+5. Aplicând exact aceeași metodă, el determină soluțiile pentru scrierea lui 12. Câte soluții de forma 3+... există?
- a. 7 b. 1 c. 2 d. 4
(Bacalaureat – Sesiunea iunie-iulie 2004)
16. Pentru problema anterioară, care este a opta soluție determinată?
- a. 2+3+7 b. 3+4+5 c. 1+5+6 d. 1+11
(Bacalaureat – Sesiunea iunie-iulie 2004)
17. Pentru problema anterioară, stabiliți care este soluția cu proprietatea că imediat după ea este generată soluția 3+4+5?
- a. 3+4+6 b. 2+3+7 c. 2+10 d. 4+8
(Bacalaureat – Sesiunea iunie-iulie 2004)
18. Se generează toate permutările distincte de 4 obiecte, numerotate de la 1 la 4, având proprietatea că 1 nu este vecin cu 3 (1 nu se află imediat după 3 și nici 3 imediat după 1) și 2 nu este vecin cu 4. Câte astfel de permutări se generează?
- a. 12 b. 24 c. 6 d. 8
(Bacalaureat – Sesiunea august 2004)
19. Se generează toate submulțimile formate din două elemente ale mulțimii {5,6,7,8}, în ordinea: 5 6, 5 7, 5 8, 6 7, 6 8, și 7 8. Dacă se utilizează exact aceeași metodă pentru a genera submulțimile de trei elemente ale mulțimii {2,3,4,5,6}, atunci penultima mulțime generată este:
- a. 3 4 5 b. 3 5 6 c. 4 5 6 d. 2 5 6
(Bacalaureat – Simulare 2006)

Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra **în echipă**. Fiecare miniproiect va conține:

- rezolvarea problemei prin implementarea metodei backtracking sau divide et impera;
 - explicarea noțiunilor teoretice folosite pentru realizarea programului;
 - justificarea metodei alese.
1. O țevă cu lungimea L trebuie să se confecționeze din n bucăți de țevă, fiecare bucată având lungimea a_i . O bucată de țevă nu poate fi tăiată. Să se găsească, dacă există, soluția de a obține țeava de lungime L prin sudarea unor bucăți de țevă de

lungimea a_i . Se citesc, dintr-un fișier text, de pe primul rând, lungimea țevii – L , și numărul de bucăți de țevă – n , și, de pe următorul rând, lungimile fiecărei bucăți de țevă. Numerele de pe un rând sunt separate prin spațiu.

2. O bară are lungimea L . Se consideră n repere de lungimi diferite. Să se genereze toate posibilitățile de a tăia bara după reperele existente, fără să rămână rest la tăiere, și fiecare reper să fie folosit cel puțin o dată. Se citesc, dintr-un fișier text de pe primul rând, lungimea barei – L , și numărul de repere – n , și de pe următorul rând, reperele. Numerele de pe un rând sunt separate prin spațiu.
 3. Să se genereze toate posibilitățile de plată a unei sume cu bancnote cu valori date, pentru fiecare valoare de bancnotă având la dispoziție un anumit număr de bancnote.
 4. Se dau n paralelipede, având fiecare dimensiunile a_i , b_i și c_i . Să se construiască un turn de înălțime maximă, prin suprapunerea acestor paralelipede. Un paralelipiped poate fi așezat pe orice față. Un paralelipiped poate fi aranjat peste un alt paralelipiped numai dacă fața lui nu depășește fața paralelipipedului pe care este pus.
 5. Să se determine numărul de partiții distincte ale unui număr natural n .
Indicație. Se descompune numărul n în două numere, n_1 și n_2 , după care fiecare număr nou obținut se descompune în alte două numere. Procesul de descompunere continuă până când $n_1 < n_2$. Numărul de partiții este egal cu numărul de descompuneri obținute.
- ```

graph TD
 4 --> 3_1["3+1"]
 4 --> 2_2["2+2"]
 3_1 --> 2_1["2+1"]
 3_1 --> 1_1["1+1"]
 2_1 --> 1_1

```
6. Într-un vector sunt memorate mai multe numere întegi. Să se plieze repetat vectorul, până când rămâne un singur element, și să se afișeze valoarea acestui element. **Plierea unui vector** înseamnă suprapunerea unei jumătăți (numită donatoare) peste cealaltă jumătate (numită receptoare). Dacă numărul de elemente ale vectorului este impar, elementul din mijloc se elimină. De exemplu, pentru vectorul  $\{1,2,3,4,5,6,7\}$ , considerând prima jumătate ca fiind cea receptoare, vectorul obținut după prima pliere este  $\{5,6,7\}$ .
  7. Într-un restaurant un meniu este format din trei feluri de mâncare. Există patru preparate culinare pentru felul unu, cinci preparate culinare pentru felul doi și trei preparate culinare pentru felul 3. Fiecare preparat culinar are un preț și un număr de calorii. Să se genereze toate meniurile pe care le poate comanda o persoană, care să nu depășească suma  $s$  și numărul de calorii  $c$ . Datele se citesc dintr-un fișier text, astfel: de pe primul rând, suma  $s$  și numărul de calorii, de pe rândul următor, în ordine, prețul fiecărui preparat culinar, și, de pe ultimul rând, în aceeași ordine, kaloriile fiecărui preparat culinar.
  8. Două persoane își împart  $n$  obiecte, astfel: prima persoană ia  $m$  obiecte, iar cealaltă persoană – restul obiectelor. Fiecare obiect are o valoare  $v_i$ . Să se genereze toate posibilitățile de distribuire a celor  $n$  obiecte între cele două persoane – astfel încât fiecare persoană să primească obiecte care să aibă aceeași valoare totală.
  9. Se citesc de la tastatură un număr natural  $n$  ( $0 < n \leq 10$ ) și apoi  $n$  numere naturale:  $a_1, a_2, a_3, \dots, a_n$ . Să se afișeze toate posibilitățile de a intercala, între toate cele  $n$  numere operatorii  $+$  și  $-$ , astfel încât, evaluând de la stânga la dreapta expresia obținută, la fiecare pas, rezultatul obținut să fie strict pozitiv.

## 2. Implementarea structurilor de date

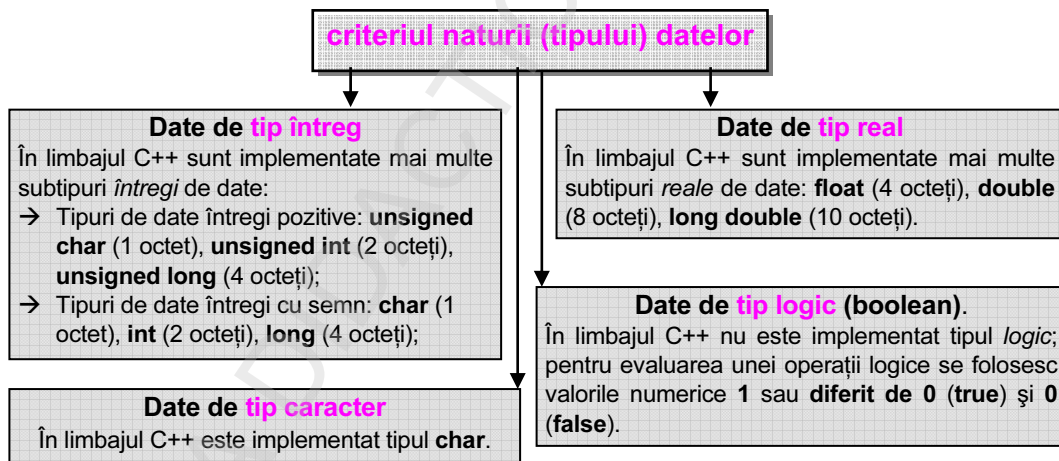
### 2.1. Datele prelucrate de algoritmi

Ați aflat deja că orice rezolvare de problemă începe prin definirea datelor, continuă cu prelucrarea lor (de exemplu, atribuirea de valori sau efectuarea unor calcule numerice) și se termină fie cu afișarea valorii lor, fie cu stocarea lor pe un mediu de memorare, în vederea unei prelucrări ulterioare. Așadar:

**Datele** sunt șiruri de biți care sunt prelucrate de calculator.

Data este o resursă la dispoziția programatorului. Orice limbaj de programare permite folosirea mai multor **tipuri de date**. Indiferent de tipul de date ales, reprezentarea sa în memoria calculatorului (fie internă, fie externă) se face printr-un șir de biți. Pentru a realiza această reprezentare, sunt implementați **algoritmi de codificare** ce asigură corespondența dintre tipul de dată și șirul de biți, atât la scrierea datelor, cât și la citirea lor. Tipul de dată ales de către programator influențează calitatea programului, deoarece el determină dimensiunea zonei de memorie alocate, algoritmul de codificare și operatorii admiși pentru prelucrare.

**Datele pot fi clasificate folosind mai multe criterii:**



**Tipul datelor** determină modul în care sunt reprezentate datele în memoria internă – și operatorii permiși pentru prelucrarea acestor date. Pentru fiecare tip de dată, sunt implementați algoritmi pentru încărcarea valorii datei în zona de memorie, algoritmi pentru adresarea zonei de memorie alocate, algoritmi pentru extragerea valorii datei din zona de memorie alocată etc. De exemplu, tipurile **unsigned char** și **unsigned int** se folosesc amândouă pentru reprezentarea numerelor întregi pozitive. Numai că, tipul **unsigned char** se memorează într-un octet și permite memorarea unor valori pozitive cuprinse între 0 și 255 ( $255=2^8-1$ ), iar tipul **unsigned int** se memorează în doi octeți (cea mai frecventă implementare a limbajului C++ pe microcalculatoare) și permite memorarea unor valori



pozitive cuprinse între 0 și 65535 ( $65535=2^{16}-1$ ). Tipul **int** se reprezintă pe 2 octeți, prin mărime și semn (folosindu-se primul bit pentru semn, iar următorii 15 biți pentru reprezentarea în binar a numărului). Acest tip poate fi folosit pentru reprezentarea numerelor întregi cu semn care au o valoare cuprinsă între -32768...32767 ( $32767=2^{15}-1$ ). Tipul **float** se reprezintă pe 4 octeți în virgulă mobilă, prin reprezentarea exponentului și a mantisei (folosindu-se primul bit pentru semnul numărului, următorul bit pentru semnul exponentului, 7 biți pentru exponent și 23 de biți pentru mantisă). Acest tip poate fi folosit pentru reprezentarea numerelor reale cu 7 cifre semnificative, în domeniul  $-10^{38}...10^{38}$ .

**Se poate testa lungimea în octeți a fiecărui tip de dată folosind operatorul sizeof().** Acesta furnizează numărul de octeți folosiți pentru memorarea unei date, precizată printr-o expresie sau prin tipul datei.

Sintaxa operatorului este: **sizeof(<tip>)**, unde <tip> este tipul datei care se testează, sau **sizeof(<expresie>)**, unde <expresie> este o expresie care se evaluează.

#### Exemple:

- operatorul **sizeof(int)** va furniza rezultatul 2, deoarece reprezentarea acestui tip de dată se face pe 2 octeți.
- considerând declarațiile **int a; float b;** – operatorul **sizeof(a+b)**; va furniza rezultatul 4, deoarece rezultatul obținut în urma evaluării expresiei este de tip **float** și reprezentarea sa necesită 4 octeți.

Pe lângă aceste tipuri de date, în limbajul C++ mai sunt implementate (Anexa 1):

- **tipul pointer** și
- **tipul referință.**

**Analiza datelor** se poate face în trei moduri:

- **Conceptual** (la nivelul algoritmului pentru rezolvarea problemei). De exemplu, o dată **n** este un număr întreg pozitiv, cu valori cuprinse între 0 și 40000.
- **Logic** (la nivelul implementării în limbajul de programare). De exemplu, pentru data **n** se alege tipul **unsigned int**, care este o dată numerică de tip întreg cu valori pozitive, ce poate lua valori de la 0 la 65535 și care permite aplicarea operatorilor aritmetici și relaționali. La nivelul limbajului de programare, este necesar să fie implementați diferiți algoritmi, care să permită folosirea acestui tip de dată: algoritmi de încărcare a valorii datei în zona de memorie, algoritmi de adresare a zonei de memorie alocate, algoritmi de extragere a valorii din zona de memorie etc.
- **Fizic** (la nivelul reprezentării ei în memoria internă corespunzător modului de implementare în limbajul de programare). De exemplu, data de tip **unsigned int** este reprezentată pe doi octeți de memorie, prin codificarea în binar a valorii numerice.

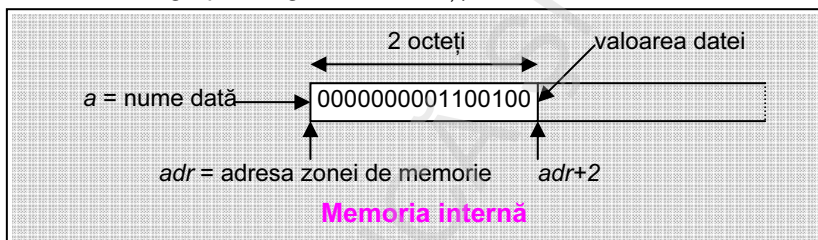
Memoria internă a calculatorului este organizată sub forma unor locații de dimensiunea unui octet, numerotate consecutiv, pornind de la 0. Aceste numere, exprimate în hexazecimal, se numesc **adrese de memorie**. Locațiile de memorie pot fi manipulate individual sau în grupuri contigue. O dată manipulată de program prin intermediul unei variabile de memorie – este caracterizată de mai multe atribute, pe care le primește în momentul în care este declarată în program.

**Exemplu:** Prin următoarea declarație a unei date în program:

```
unsigned int a=100;
```

data va primi următoarele atribute:

- **Numele.** Este identificatorul folosit pentru referirea datei în cadrul programului. În exemplu, *a*.
- **Adresa.** Este adresa de memorie internă la care se alocă spațiu datei respective, pentru a stoca valoarea datei. În exemplu, *adr* (un număr binar care reprezintă adresa unui octet de memorie). Atribuirea adresei este făcută de sistem, în funcție de locațiile cu dimensiunea de 2 octeți libere din zona de lucru alocată programului.
- **Valoarea.** Este conținutul, la un moment dat, al zonei de memorie rezervată datei. În exemplu, 100.
- **Tipul.** Determină: *domeniul de definiție intern* al datei (mulțimea în care poate lua valori data), *operatorii* care pot fi aplicați pe acea dată și modul în care data este *reprezentată în memoria internă* (metoda de codificare în binar a valorii datei). În exemplu, tipul este **unsigned int** și determină: **domeniul de definiție intern** al datei (intervalul [0, 65535]), **operatorii** care pot fi aplicați pe acea dată (operatorii permisi de tipul numeric – aritmetici, relaționali, logici, logici pe biți etc.) și modul în care data este **reprezentată în memoria internă** (reprezentarea prin conversia numărului din zecimal în binar).
- **Lungimea.** Este dimensiunea zonei de memorie alocate datei și se măsoară în octeți. Ea depinde de modul de reprezentare internă a tipului de dată. În exemplu, 2 octeți (datei i se alocă un grup contiguu de 2 octeți).

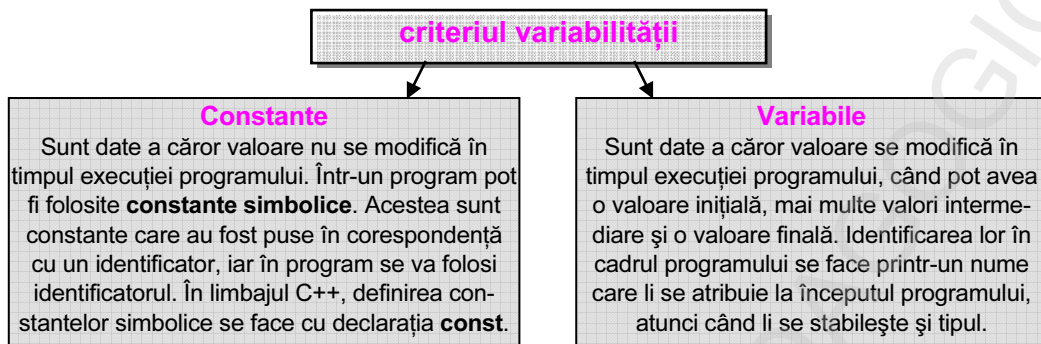


În urma declarării datelor într-un program, sistemul de operare își construiește o tabelă prin care stabilește corespondența dintre *numele* datei, *adresa* la care se memorează data și *lungimea* zonei de memorie alocată. Pentru exemplul precedent, se va memora în tabelă: numele *a*, adresa *adr* și lungimea 2. Atunci când se va cere printr-o instrucțiune din program să se afișeze valoarea acestei date, sistemul va căuta în tabelă identificatorul *a*. Dacă îl găsește, va citi conținutul zonei de memorie care începe de la adresa *adr* și are lungimea de 2 octeți, îl va converti din modul de reprezentare internă în modul de reprezentare externă și va afișa valoarea obținută în urma conversiei, la dispozitivul desemnat pentru afișarea ei. Dacă nu găsește acest identificator, sistemul va afișa un mesaj de eroare prin care va preciza că nu există această dată. Pentru a putea manipula o dată, sistemul trebuie să identifice zona de memorie în care este stocată. Identificarea acestei zone se poate face atât prin numele datei (*a*), cât și prin adresa la care este memorată (*adr*).

### Temă



1. Declarați două variabile de memorie: una de tip numeric întreg și una de tip numeric real.
2. Declarați două variabile: una de tip numeric și una de tip caracter.
3. Enumerați, pentru fiecare tip de dată, subtipurile implementate. Arătați modul în care pot fi declarate și precizați spațiul de memorie alocat fiecărui subtip și domeniul de valori.
4. Analizați data de tip **char** (reprezentare în memoria internă, încadrare în criteriile prezentate, operatori permisi etc.).



Se recomandă **folosirea constantelor simbolice** în următoarele cazuri:

- valoarea datei rămâne constantă pe o perioadă de timp, după care se poate modifica (de exemplu, într-un program pentru calcularea salariilor, impozitul se poate modifica după o anumită perioadă de timp și, folosind constante simbolice, se poate face mult mai ușor modificarea în programul sursă),
- data este o constantă matematică (de exemplu, constanta  $\pi$ ) sau
- o dată constantă este folosită în mai multe locuri în program.

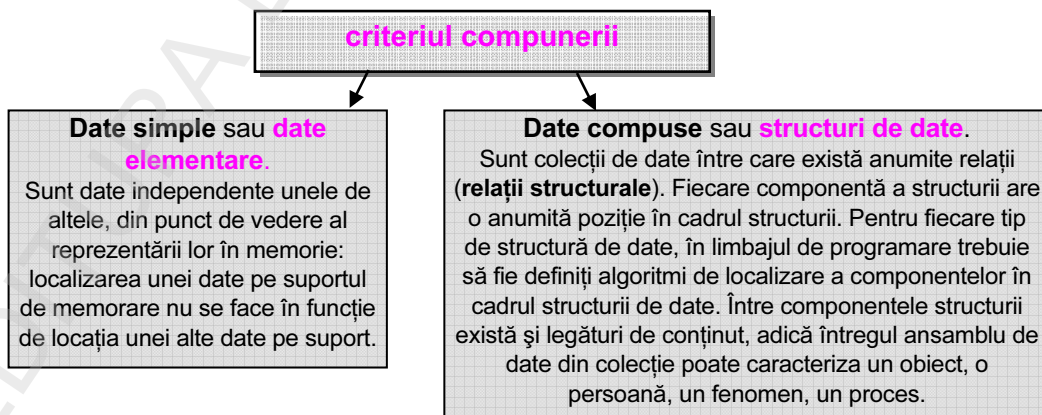
### Observații:

1. Un caracter delimitat de apostrofuri este diferit de un caracter delimitat de ghilimele (de exemplu, 'a' este diferit de "a"). Primul este o constantă de tip **char** (o dată elementară), iar al doilea este o structură de date de tip **șir de caractere** ce conține un singur caracter.
2. Există constante simbolice de sistem (definite în bibliotecile limbajului) pe care le puteți folosi în programe: **MAXINT** (cea mai mare valoare de tip **int** – întreg); **MAXSHORT** (cea mai mare valoare de tip **short** – întreg scurt); **MAXLONG** (cea mai mare valoare de tip **long** – întreg lung); **MAXFLOAT**, respectiv **MINFLOAT** (cea mai mare valoare, respectiv cea mai mică valoare de tip **float** – real simplă precizie); **MAXDOUBLE**, respectiv **MINDOUBLE** (cea mai mare valoare, respectiv cea mai mică valoare de tip **double** – real dublă precizie); și **M\_PI** pentru numărul  $\pi$ .

### Temă



1. Declarați două constante: una de tip numeric întreg și una de tip numeric real.
2. Care este valoarea următorului număr, exprimat în notație științifică: 7.54E5? Dar a numărului -1.354e-6? Ce fel de constante sunt acestea: simbolice sau literale?



Ați studiat structurile de date **implementate la nivel fizic** în limbajul C++:

- **tabloul de memorie cu o dimensiune (vectorul)** și
- **fișierul**.

Tabloul de memorie este o structură de date omogenă, internă și temporară, iar fișierul este structură de date omogenă, externă și permanentă. Pentru rezolvarea unor probleme, nu sunt suficiente numai aceste structuri de date. Limbajul C++ mai permite și folosirea următoarelor structuri fizice de date:

- **tabloul de memorie cu două dimensiuni (matricea);**
- **șirul de caractere** și
- **înregistrarea**.

### Temă



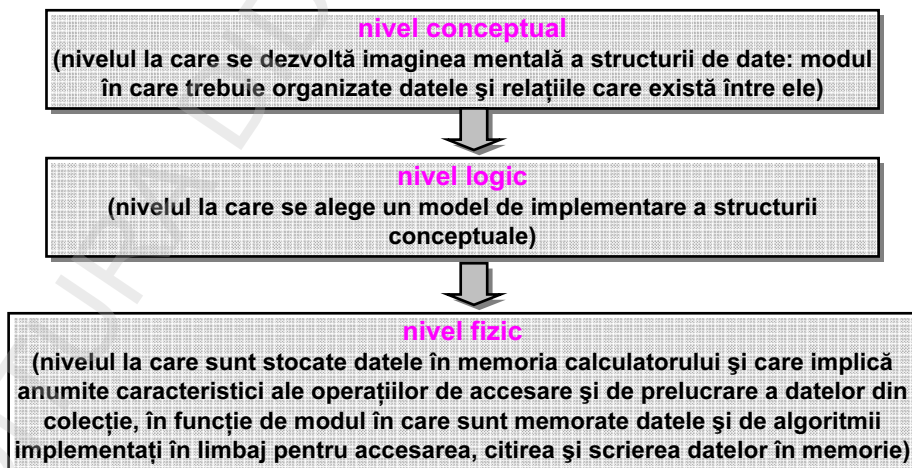
1. Structurii de date **a**, declarată cu **double a[10];**, i se alocă un spațiu de memorare continuu, începând de la adresa 200. La ce adresă se va găsi data **a[4]**?
2. Analizați, din punct de vedere al modului în care se încadrează în criteriile prezentate, următoarele tipuri de date.

```
const int MAXIM=1000, VZ[3]={0};
const float PI=3.14;
typedef unsigned char byte;
typedef enum {False,True} boolean;
int a, b[10]; float c, d[2]; char x; byte y; boolean z;
```

- a) Câte date au fost definite? Enumerați-le!
- b) Câte variabile de memorie au fost definite? Enumerați-le!
- c) Câte date structurate au fost definite? Enumerați-le!

## 2.2. Tabloul de memorie bidimensional (matricea)

Organizarea în structuri de date a datelor prelucrate de algoritmi simplifică multe dintre operațiile de prelucrare. Atunci când organizați datele într-o structură de date, trebuie să identificați modul în care puteți avea acces la ele și operațiile pe care le puteți executa cu datele din colecție. Procesul de organizare a datelor în colecții este un proces care se desfășoară pe trei niveluri care interacționează între ele, pornind de la nivelul conceptual:



## Studiu de caz

**Scop:** exemplificarea modului în care, pentru a rezolva problema, alegeți organizarea datelor într-o structură de date de tip tablou de memorie.

**Enunțul problemei 1.** O firmă de transport are un parc de 10 mașini, cu capacități de transport diferite. Trebuie să se determine câte dintre aceste mașini au cea mai mare capacitate de transport.

Pentru rezolvarea problemei, trebuie stabilită structura de date care se va folosi:

→ La nivel **conceptual** – capacitățile de transport ale mașinilor reprezintă un șir de numere întregi, aranjate într-o ordine aleatorie, în care trebuie căutat numărul cel mai mare și de câte ori apare în șir.

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 20 | 40 | 50 | 30 | 20 | 40 | 50 | 40 | 30 | 40 |
|----|----|----|----|----|----|----|----|----|----|

→ La nivel **logic** – implementarea permisă de limbajul C++ a unei colecții de date omogene este **vectorul**, fiecare număr întreg fiind un element al structurii. Pentru rezolvarea problemei se vor folosi următorii algoritmi: algoritmul pentru parcurgerea vectorului la memorarea numerelor, algoritmul pentru determinarea valorii maxime dintr-un șir de numere și algoritmul de căutare în vector a elementelor cu o valoare precizată (valoarea maximă).

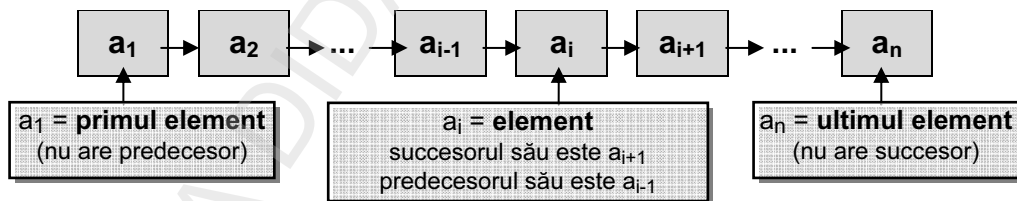
```
int a[10];
```

→ La nivel **fizic** – numerele vor fi memorate într-o zonă continuă de memorie internă, fiecărui număr alocându-i-se același spațiu pentru memorare. Identificarea unui element al structurii se face prin numărul său de ordine (indicele).

|      |      |      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|------|------|
| 20   | 40   | 50   | 30   | 20   | 40   | 50   | 40   | 30   | 40   |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] |

Dacă se dorește păstrarea datelor memorate (capacitățile de transport ale mașinilor din parcul auto) pentru prelucrarea lor ulterior, se vor salva într-un fișier text, de unde vor fi restaurate în memoria internă într-un vector, la fiecare prelucrare (execuție a programului).

**Observație.** În structura de date folosită (vectorul), între elementele structurii există o relație de ordine în care fiecare element are un succesor și un predecesor. Acest tip de structură este o **structură liniară**.

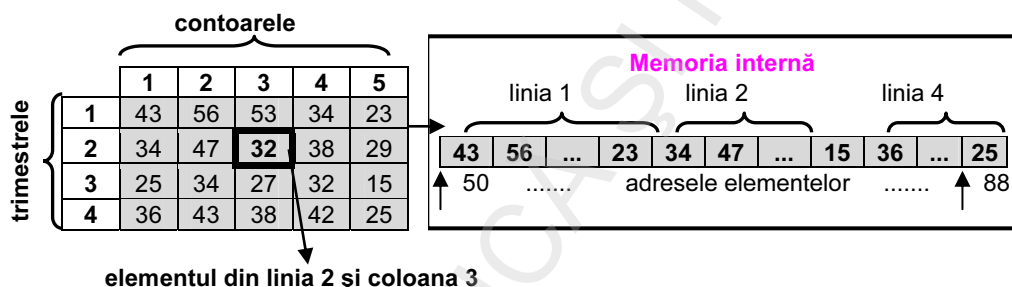


**Enunțul problemei 2.** Sunt cinci contoare pentru energia electrică, ce se citesc trimestrial, și se înregistrează într-un tabel consumul trimestrial (în mii de kWh). Trebuie să se determine: media consumului trimestrial pe un singur contor și pe toate contoarele, cel mai mare consum trimestrial, cel mai mic consum trimestrial, contorul cu cel mai mic consum și contorul cu cel mai mare consum.

Pentru rezolvarea problemei, trebuie stabilită structura de date care se va folosi:

→ La nivel **conceptual** – contoarele sunt organizate într-un tabel cu patru linii și cinci coloane în care sunt memorate numere întregi. Tabelul va avea 4 linii, corespunzătoare celor 4 trimestre, și 5 coloane, corespunzătoare celor cinci contoare.

- La nivel **logic** – implementarea permisă de limbajul C++ a unei colecții de date omogene este tabloul de memorie. Pentru rezolvarea problemei se vor folosi următorii algoritmi: algoritmul pentru parcurgerea tabloului de memorie, pentru memorarea numerelor, algoritmul pentru determinarea mediei aritmetice și algoritmi de determinare a valorii minime, respectiv maxime. Dacă pentru reprezentarea datelor s-ar folosi tabloul de memorie cu o singură dimensiune (vectorul), algoritmi de prelucrare ar fi foarte complicați. Soluția, în acest caz, este un tablou bidimensional (**matrice**) cu patru linii și cinci coloane. Identificarea unui element din tablou se va face de data aceasta nu printr-un număr de ordine, ci prin numărul liniei și numărul coloanei. Dacă numele tabloului bidimensional este **b**, elementul **b<sub>2,3</sub>** este elementul din linia 2 și coloana 3 și, în exemplu, are valoarea 32.
- La nivel **fizic** – memoria internă nu are o structură matriceală. Ea este formată din locații de memorare care au adrese consecutive. Din această cauză, structura dreptunghiulară a tabelului trebuie simulată. Spațiul de memorare alocat tabelului va fi o zonă continuă, de 40 de octeți, în care se vor memora datele din tabel, linie cu linie. Numărul de ordine **m** al elementului din linia **i** și coloana **j** dintr-un tablou cu **n** coloane este:
- $$m = n \times (i-1) + j$$



Și în cazul acestui exemplu, dacă se dorește păstrarea datelor memorate (valorile consumului trimestrial al fiecărui contor) pentru prelucrarea lor ulterior, se vor salva într-un fișier text, de unde vor fi restaurate în memoria internă într-un tablou bidimensional, la fiecare prelucrare (execuție a programului).

### Temă



Rezolvați cele două probleme folosind structura de date de tipul tablou de memorie cu o dimensiune.

**Observație.** Metoda de memorare a elementelor tabloului, în ordine, unul după altul, într-o zonă continuă de memorie, în locații cu aceeași dimensiune, este folosită pentru identificarea elementelor structurii și se realizează printr-un ansamblu de indici. Numărul de indici (**k**) folosit pentru identificarea elementelor determină **dimensiunea** tabloului. Astfel:

- pentru  $k=1$ , structura este un tablou unidimensional numit și **vector**;
- pentru  $k=2$ , structura este un tablou bidimensional numit și **matrice**.

În primul exemplu, s-a folosit un tablou de memorie cu numele **a**, cu o singură dimensiune și cu 10 elemente. În al doilea exemplu, s-a folosit tabloul cu numele **b**, cu două dimensiuni, cu 4 elemente pe o dimensiune (4 linii) și 5 elemente pe cealaltă dimensiune (5 coloane). În cel de al doilea caz, tabloul are tot 20 de elemente. Chiar dacă, din punct de vedere al memoriei interne, spațiul de memorare alocat este tot sub forma unui bloc continuu de 20, respectiv 40 de octeți, cele două tipuri de tablouri (cu o dimensiune sau cu două dimensiuni) diferă între ele prin modul în care pot fi identificate elementele structurii. Identificarea unui element al tabloului se face prin numele tabloului și valorile indicilor după



toate dimensiunile. De exemplu,  $a_6$  înseamnă elementul 6 din tabloul  $a$ , iar  $b_{2,4}$  înseamnă elementul de pe linia 2 și coloana 4 din tabloul  $b$ .

Într-o matrice cu  $m$  linii și  $n$  coloane, numărul de ordine al elementului de pe linia  $n1$  și din coloana  $nc$  este:

→ Dacă înregistrarea în memorie se face **linie cu linie**:  $nr\_ord\_l = (n1-1) \times n + nc$ .

→ Dacă înregistrarea în memorie se face **coloană cu coloană**:  $nr\_ord\_c = (nc-1) \times m + n1$ .

Dacă matricea se înregistrează începând de la adresa  $adr$ , iar un element ocupă o zonă de memorie cu dimensiunea  $dim$ , adresa la care se găsește elementul care are numărul de ordine  $nr\_ord$  este egală cu  $adr + dim \times (nr\_ord - 1)$ .

O structură de date care, la **nivel conceptual**, are imaginea **unui tabel care conține date de același tip**, se implementează la nivel logic cu ajutorul unui **tablou bidimensional**.

### 2.2.1. Implementarea tabloului bidimensional în C++

Declararea unui tablou de memorie de tip matrice (cu două dimensiuni) se face prin instrucțiunea:

```
<tip_dată> <nume> [<nr_1>] [<nr_2>];
```

unde **<tip\_dată>** precizează tipul elementelor matricei, **<nume>** este identificatorul matricei, iar **<nr\_1>** și **<nr\_2>** sunt două constante întregi care specifică numărul de elemente ale matricei pentru fiecare dimensiune: **<nr\_1>** – numărul de linii, iar **<nr\_2>** – numărul de coloane. De exemplu, prin instrucțiunile declarative:

```
int a[2][4];
```

```
float b[3][5];
```

se declară două matrice: **a** cu 8 elemente de tip **int**, care are 2 linii și 4 coloane, și **b**, cu 15 elemente de tip **float**, care are 3 linii și 5 coloane.

Și la declararea unei matrice se pot atribui valori inițiale elementelor, cu instrucțiunea:

```
<tip_dată> <nume> [<nr_1>] [<nr_2>] = {listă_valori};
```

unde valorile din listă se vor atribui elementelor ținând cont de faptul că memorarea lor în zona alocată matricei se face linie după linie. De exemplu, prin instrucțiunea declarativă:

```
int a[2][4] = {1, 2, 3, 4, 5, 6, 7, 8}; sau
```

```
int a[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

se declară o matrice **a** cu 8 elemente de tip **int**, care are 2 linii și 4 coloane în care s-au atribuit valori inițiale elementelor.

Pentru prelucrarea datelor memorate într-o matrice, **referirea la un element al matricei** se face prin construcția:

```
<nume> [<indice_1>] [<indice_2>]
```

unde **nume** este numele matricei, **<indice\_1>** este numărul de ordine al liniei, iar **<indice\_2>** este numărul de ordine al coloanei.

|                          |   |                    |   |   |   |
|--------------------------|---|--------------------|---|---|---|
| indicele coloanei        | j | →                  |   |   |   |
|                          |   | 0                  | 1 | 2 | 3 |
| indicele liniei          | i | 0                  | 1 | 2 | 3 |
| element = <b>a[i][j]</b> | ↓ | 1                  | 2 | 3 | 4 |
|                          |   | 5                  | 6 | 7 | 8 |
|                          |   | <b>a[1][2] = 7</b> |   |   |   |



**Atenție**

Deoarece adresa matricei este și adresa primului element, iar indicii reprezintă deplasarea elementului față de adresa matricei, numerotarea indicilor se face pornind de la 0, iar  $0 \leq \text{indice\_1} < m$  și  $0 \leq \text{indice\_2} < n$ , unde  $m$  reprezintă numărul de linii, și  $n$  numărul de coloane ale matricei, precizate la declararea ei. În exemplu,  $a[1][2]$  reprezintă elementul din linia a 2-a și coloana a 3-a din matricea  $a$ , și are valoarea 7.

**Atenție**

Tabloul de memorie fiind o structură de date statică, la declararea lui i se alocă o zonă fixă de memorie. **Lungimea tabloului de memorie** reprezintă **numărul de elemente** ale tabloului. La declararea tabloului, este posibil să nu se cunoască numărul de elemente care vor fi prelucrate la fiecare execuție a programului. În prelucrarea tabloului de memorie se folosesc două lungimi:

- **Lungimea fizică.** Reprezintă numărul de elemente stabilit la declararea tabloului. Este numărul maxim de elemente care pot fi stocate în spațiul de memorie rezervat tabloului. În cazul unei matrice, lungimea fizică este dată de produsul dintre numărul de linii și numărul de coloane precizate la declararea matricei.
- **Lungimea logică.** Reprezintă numărul de elemente care vor fi prelucrate la execuția programului. Este mai mic sau cel mult egal cu lungimea fizică (numărul maxim de elemente). Lungimea logică se comunică în timpul execuției programului (de la tastatură, sau se citește dintr-un fișier text). În cazul matricei, ea reprezintă produsul dintre numărul de linii și numărul de coloane ale matricei ( $m \times n$ ).

Pentru elementul  $a[i][j]$ , unde  $i$  și  $j$  reprezintă indicele liniei, respectiv al coloanei, se definesc **succesorul** elementului și **predecesorul** lui, astfel:

$$\text{succ}(a[i][j]) = \begin{cases} a[i+1][0] & \text{dacă } j=n-1 \\ a[i][j+1] & \text{dacă } j \neq n-1 \end{cases}$$

$$\text{pred}(a[i][j]) = \begin{cases} a[i-1][n-1] & \text{dacă } j=0 \\ a[i][j-1] & \text{dacă } j \neq 0 \end{cases}$$

Elementul  $a[0][0]$  nu are predecesor, iar elementul  $a[m-1][n-1]$  nu are succesor.

## 2.2.2. Algoritmi pentru prelucrarea tablourilor bidimensionale

### 2.2.2.1. Algoritmi pentru parcurgerea elementelor unei matrice

**Parcurgerea elementelor unei matrice** se poate face:

- de la primul element până la ultimul element, sau
- de la ultimul element până la primul element.

Secvența de instrucțiuni pentru **parcurgerea elementelor unei matrice de la primul element la ultimul element** este:

```
int i,j,m,n,a[10][10];
// se declară variabilele i și j pentru indicele elementului,
// i - numărul liniei și j - numărul coloanei
// m și n pentru lungimea logică a matricei (m - numărul de
// linii și n - numărul de coloane) și a pentru o matrice cu
// lungimea fizică de 10 linii și 10 coloane.
cout<<"n= "; cin>>m; // se citește numărul de linii ale matricei
cout<<"m= "; cin>>n; // se citește numărul de coloane ale matricei
for (i=0;i<m;i++) // se parcurg liniile matricei
 for (j=0;j<n;j++) //se parcurg coloanele matricei de pe o linie
 ; //instrucțiunea pentru prelucrarea elementului a[i][j]
```

Secvența de instrucțiuni pentru **parcurea elementelor unei matrice de la ultimul element la primul element** este:

```
int i,j,m,n,a[10][10];
cout<<"m= "; cin>>m; cout<<"n= "; cin>>n;
for (i=m-1;i>=0;i--) // se parcurg liniile matricei
 for (j=n-1;j>=0;j--) //se parcurg coloanele matricei de pe o linie
 ; //instrucțiunea pentru prelucrarea elementului a[i][j]
```

**Exemplul 1 – Citirea de la tastatură a valorilor elementelor unei matrice.** Secvența de instrucțiuni este:

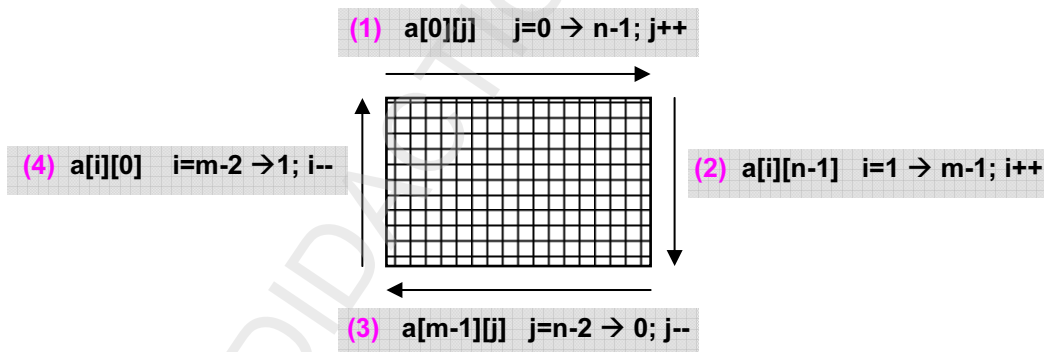
```
int i,j,m,n,a[10][10];
cout<<"m= "; cin>>m; cout<<"n= "; cin>>n;
for (i=0;i<m;i++)
 for (j=0;j<n;j++)
 {cout<<"a("<<i+1<<","<<j+1<<")= "; cin>>a[i][j];}
..... //se prelucrează elementele matricei
```

**Exemplul 2 – Afișarea pe ecran a valorilor elementelor unei matrice.** Secvența de instrucțiuni este:

```
int i,j,m,n,a[10][10];
..... //se prelucrează elementele matricei
for (i=0;i<m;i++)
 {for (j=0;j<n;j++) cout<<a[i][j]<<" ";
 cout<<endl;}
```

### Parcurea unei matrice pe contur:

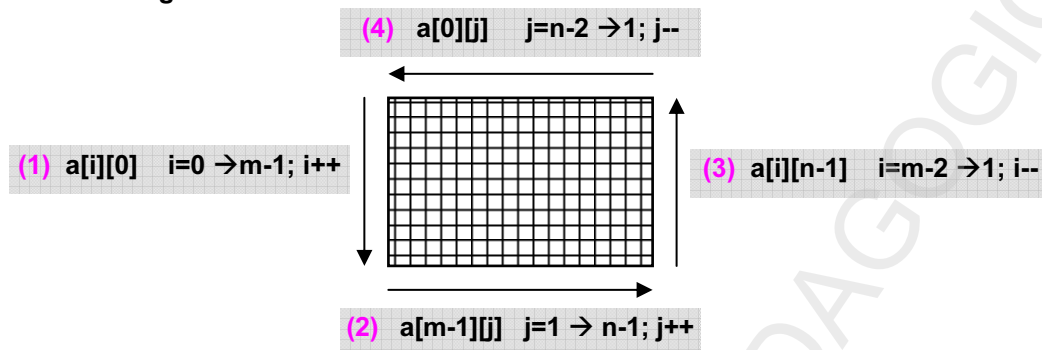
→ în sensul acelor de ceasornic (în sens invers trigonometric).



Secvența de instrucțiuni este:

```
int i,j,m,n,a[10][10];
..... //se citesc valorile elementelor matricei
for (j=0;j<n;j++) cout<<a[0][j]<<" ";
for (i=1;i<m;i++) cout<<a[i][n-1]<<" ";
for (j=n-2;j>=0;j--) cout<<a[m-1][j]<<" ";
for (i=m-2;i>0;i--) cout<<a[i][0]<<" ";
```

→ în sens trigonometric.



### Studiu de caz

**Scop:** exemplificarea modului în care se aplică algoritmii pentru parcurgerea tablourilor de memorie bidimensionale.

**Enunțul problemei 1:** Să se inverseze coloanele unei matrice cu  $n$  linii și  $m$  coloane ( $n \leq 10, m \leq 10$ ) cu elementele numere întregi.

Se folosește algoritmul de inversare a unui vector în el însuși, pentru fiecare linie a matricei.

```
#include <iostream.h>
void main()
{int n,m,i,j,x,a[10][10];
 cout<<"n ="; cin>>n; cout<<"m ="; cin>>m;
 for (i=0;i<n;i++)
 for (j=0;j<m;j++)
 {cout<<"a["<<i+1<<","<<j+1<<"]=" "; cin>>a[i][j];}
 for (i=0;i<n;i++)
 for (j=0;j<m/2;j++)
 {x=a[i][j]; a[i][j]=a[i][m-j-1]; a[i][m-j-1]=x;}
 for (i=0;i<n;i++)
 {for (j=0;j<m;j++) cout<<a[i][j]<<" ";
 cout<<endl; }}
```

**Enunțul problemei 2:** Se citesc de la tastatură elementele de tip întreg ale unei matrice cu  $m$  linii și  $n$  coloane. Să se calculeze într-un vector suma elementelor din fiecare linie a matricei. Problema se va descompune în subprobleme și fiecare subproblemă va fi implementată cu ajutorul unui subprogram.

Se folosesc subprogramele: **cit\_mat()** pentru a citi elementele matricei, **suma()** pentru a calcula suma elementelor matricei de pe fiecare linie și **afis\_vec()** pentru a afișa elementele vectorului.

```
#include <iostream.h>
void cit_mat(int a[][50] , int &m, int &n)
{cout<<"m= "; cin>>m; cout<<"n= "; cin>>n;
 for (int i=0;i<m;i++)
 for (int j=0;j<n;j++)
 {cout<<"a["<<i+1<<","<<j+1<<"]=" "; cin>>a[i][j];}}
void afis_vec(int b[], int m)
{for (int i=0;i<m;i++) cout<<b[i]<<" ";
```

```

void suma(int a[][50], int b[], int m, int n)
{int i,j;
 for (i=0;i<m;i++) b[i]=0;
 for (i=0;i<m;i++)
 for (j=0;j<n;j++) b[i]+=a[i][j];}

void main()
{int a[50][50],b[50],m,n;
 cit_mat(a,m,n); suma(a,b,m,n); afis_vec(b,m);}

```

### Atenție

Atunci când transmiteți o matrice ca parametru, la declararea parametrului nu trebuie să precizați numărul de rânduri ale matricei, dar trebuie să precizați numărul de coloane.

**Enunțul problemei 3:** Se citesc de la tastatură elementele unei matrice cu  $m$  linii și  $n$  coloane. Elementele matricei sunt de tip întreg. Să se afișeze maximul dintre elementele matricei și poziția primei apariții a acestuia (linia și coloana). Problema se va descompune în subprobleme și fiecare subproblemă va fi implementată cu ajutorul unui subprogram.

Se folosesc următoarele subprograme: **citeste()** pentru a citi elementele matricei și **maxim()** pentru a obține informațiile despre valoarea maximă.

```

#include <iostream.h>

void citeste(int a[][50], int &m, int &n)
{cout<<"m= "; cin>>m; cout<<"n= "; cin>>n;
 for (int i=0;i<m;i++)
 for (int j=0;j<n;j++)
 {cout<<"a["<<i+1<<","<<j+1<<"]=" "; cin>>a[i][j];}}

void maxim(int a[][50], int m, int n, int &max, int &nl, int &nc)
{max=a[0][0]; nl=0; nc=0;
 for (int i=0;i<m;i++)
 for (int j=0;j<n;j++)
 if (a[i][j]>max) {max=a[i][j]; nl=i; nc=j;}}

void main()
{int a[50][50],m,n,max,nl,nc; citeste(a,m,n);
 maxim(a,m,n,max,nl,nc);
 cout<<"maximul este "<<max<<"si apare prima data "<<endl;
 cout<<" in linia "<<nl+1<<" si coloana "<<nc+1<<endl; }

```

**Enunțul problemei 4:** Se consideră o matrice  $a$  cu  $m$  linii și  $n$  coloane, cu elemente numere reale. Valorile pentru  $m$  și  $n$  și pentru elementele matricei se citesc dintr-un fișier text, unde sunt scrise astfel: pe primul rând valorile pentru  $m$  și  $n$ , iar pe următoarele  $m$  rânduri, câte  $n$  elemente de pe fiecare linie a matricei. Numerele sunt separate prin spațiu. Să se bordeze matricea cu coloana  $n+1$ , ale cărei elemente  $a[i][n+1]$  au ca valoare media aritmetică a celor  $n$  elemente din linia  $i$ , și cu linia  $m+1$ , ale cărei elemente  $a[m+1][j]$  au ca valoare media aritmetică a celor  $n$  elemente din coloana  $j$ . Să se scrie într-un fișier text matricea obținută. Problema se va descompune în subprobleme – și fiecare subproblemă va fi implementată cu ajutorul unui subprogram.

Se folosesc următoarele subprograme: **citeste()** pentru a citi matricea inițială din fișierul text, **bordeaza()** pentru a borda matricea și **scrie()** pentru a scrie într-un fișier text matricea obținută.

```
#include <fstream.h>
fstream f1("mat1.txt",ios::in),f2("mat2.txt",ios::out);

void citeste(float a[][10], int &m, int &n)
{f1>>m>>n;
 for (int i=0;i<m;i++)
 for (int j=0;j<n;j++) f1>>a[i][j];
 f1.close();}

void scrie(float a[][10], int m, int n)
{f2<<m<<" "<<n<<endl;
 for (int i=0;i<m;i++)
 {for (int j=0;j<n;j++) f2<<a[i][j]<<" ";
 f2<<endl;}
 f2.close();}

void bordeaza(float a[][10], int &m, int &n)
{int i,j; float s;
 for (i=0;i<m;i++)
 {for (j=0,s=0;j<n;j++) s+=a[i][j];
 a[i][n]=s/n;}
 n++;
 for (i=0;i<n;i++)
 {for (j=0,s=0;j<m;j++) s+=a[j][i];
 a[m][i]=s/m;}
 m++;}

void main()
{int n,m; float a[10][10];
 citeste(a,m,n); bordeaza(a,m,n); scrie(a,m,n); }
```

**Enunțul problemei 4:** Se consideră o matrice **a** cu **m** linii și **n** coloane, cu elemente numere reale. Valorile pentru **m** și **n** și pentru elementele matricei se citesc de la tastatură. Să se calculeze suma elementelor pare din matrice, folosind o funcție recursivă.

Funcția recursivă este **suma(i, j, n, a)**, unde **a** este matricea, **i** și **j** indicii folosiți pentru a identifica un element al matricei și **n** numărul de coloane. Cu indicele **i** se parcurg liniile de la ultima până la prima (indicele **i** ia valori de la **m-1** până la 0), iar cu indicele **j** se parcurg coloanele de la ultima până la prima (indicele **j** ia valori de la **n-1** până la 0). Se evaluează funcția **suma(m-1, n-1, n, a)**.

```
#include <iostream.h>

void citeste (int a[][20], int &m, int &n)
{int i,j; cout<<"m= "; cin>>m; cout<<"n= "; cin>>n;
 for (i=0;i<m;i++)
 for (j=0;j<n;j++)
 {cout<<"a["<<i+1<<","<<j+1<<"]=" "; cin>>a[i][j];}

int suma(int i,int j,int n, int a[][20])
{if(i==0 && j==0)
 if (a[0][0]%2==0) return a[0][0];
 else return 0;
```

```

else if (a[i][j]%2==0)
 if (j==0) return a[i][j]+suma(i-1,n-1,n,a);
 else return a[i][j]+suma(i,j-1,n,a);
else
 if (j==0) return suma(i-1,n-1,n,a);
 else return suma(i,j-1,n,a);
}

void main()
{int m,n,a[20][20]; citeste(a,m,n);
cout<<"suma este "<<suma(m-1,n-1,n,a)<<endl; }

```



### Temă



1. Să se verifice că o matrice pătrată cu  $m$  linii și  $n$  coloane ( $m$  și  $n$  și elementele matricei se citesc dintr-un fișier text) este matricea zero (matricea care are toate elementele egale cu 0).
2. Să se afișeze elementele de pe conturul matricei, parcurgerea lor făcându-se în sens trigonometric.
3. Pentru o valoare  $n$  (număr natural de cel mult o cifră) citită de la tastatură se cere să se scrie un program care construiește în memorie o matrice de  $n$  linii și  $n$  coloane formată numai din elemente de 1 și 2, elementele aflate pe cele 4 margini ale tabloului fiind egale cu 1, cele din interior fiind egale cu 2. Elementele matricei se scriu pe ecran, pe linii, ca în exemplul următor. Pentru  $n=4$ , se afișează matricea prezentată alăturat.

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 1 |
| 1 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 |

(Bacalaureat – Simulare 2006)

Următoarele probleme se vor descompune în subprobleme și fiecare subproblemă va fi implementată cu ajutorul unui **subprogram**.

4. Se consideră o matrice  $a$  cu  $m$  linii și  $n$  coloane, cu elemente numere reale. Valorile pentru  $n$  și  $m$  și elementele matricei se citesc de la tastatură. Să se afișeze numărul liniei și numărul coloanei pe care suma elementelor este maximă.
5. Se consideră o matrice  $a$  cu  $m$  linii și  $n$  coloane, cu elemente numere reale. Valorile pentru  $m$  și  $n$  și elementele matricei se citesc de la tastatură. Se mai citesc de la tastatură două numere întregi,  $p$  ( $1 \leq p \leq n$ ) și  $q$  ( $1 \leq q \leq n$ ). Să se interschimbe coloanele  $p$  și  $q$  ale matricei. Să se afișeze matricea obținută.
6. Se consideră o matrice  $a$  cu  $m$  linii și  $n$  coloane, cu elemente numere reale. Valorile pentru  $n$  și  $m$  și elementele matricei se citesc dintr-un fișier text. Se citesc de la tastatură două numere întregi,  $p$  ( $1 \leq p \leq m$ ) și  $q$  ( $1 \leq q \leq m$ ). Să se interschimbe liniile  $p$  și  $q$  ale matricei. Să se afișeze matricea obținută.
7. Se consideră o matrice  $a$  cu  $m$  linii și  $n$  coloane, cu elemente numere reale. Valorile pentru  $n$  și  $m$  și elementele matricei se citesc dintr-un fișier text. Se citesc de la tastatură două numere întregi,  $p$  ( $1 \leq p \leq m$ ) și  $q$  ( $1 \leq q \leq n$ ). Să se elimine din matrice linia  $p$  și coloana  $q$ . Să se scrie într-un fișier matricea obținută.
8. Se citesc de la tastatură două numere naturale,  $m$  și  $n$ , care reprezintă dimensiunile unei matrice cu numere întregi, și elementele matricei. Afișați liniile și coloanele matricei care au numai numere pare.
9. Se citesc de la tastatură două numere naturale,  $m$  și  $n$ , care reprezintă dimensiunile unei matrice cu numere întregi, și elementele matricei. Afișați liniile și coloanele matricei care au elementele aranjate crescător.

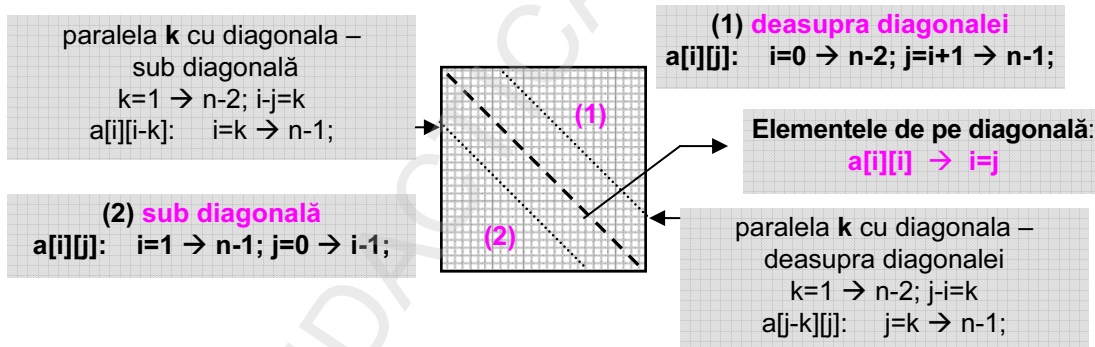
10. Pentru o matrice cu  $n$  linii și  $m$  coloane, scrieți un subprogram recursiv care să inverseze linia  $p$  cu linia  $q$  – și un subprogram recursiv care să inverseze coloana  $x$  cu coloana  $y$ .
11. Se consideră o matrice  $a$  cu numere întregi, cu dimensiunea  $m$  linii și  $n$  coloane, și un vector  $b$ , cu  $m \times n$  elemente. Se citesc dintr-un fișier text cele două numere,  $m$  și  $n$ , și elementele matricei. Să se copieze în vectorul  $b$  elementele matricei  $a$ , parcurse:
- linie cu linie,
  - coloană cu coloană,
  - în spirală (pornind de la elementul  $a[0][0]$  în sens invers trigonometric).
12. Se consideră o matrice  $a$  cu numere întregi, cu dimensiunea  $m$  linii și  $n$  coloane. Folosind metoda **divide et impera**:
- să se interschimbe coloana  $p$  cu coloana  $q$  ( $p$  și  $q$  se citesc de la tastatură);
  - să se determine simultan valoarea minimă și valoarea maximă din matrice.

### 2.2.2.2. Algoritmi pentru prelucrarea matricelor pătrate

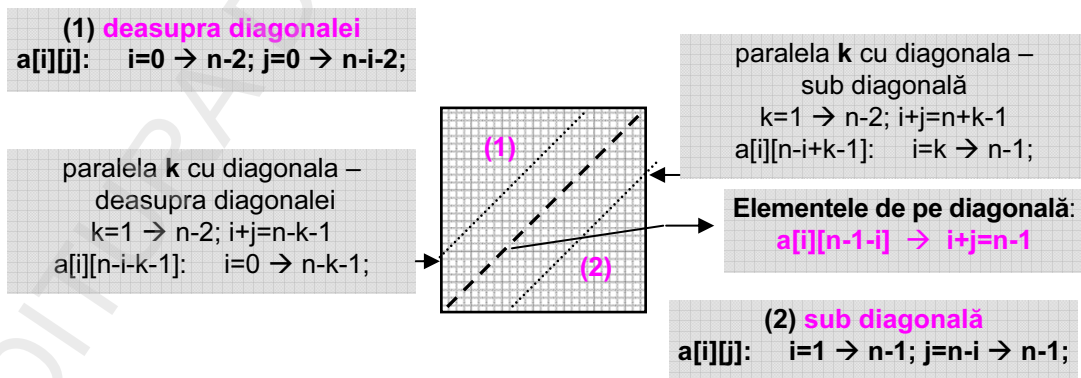
O **matrice pătrată** este o matrice în care numărul liniilor este egal cu numărul coloanelor. Lungimea logică a unei matrice pătrate este  $n \times n$  (unde  $n$  = numărul de linii și de coloane). Ea este împărțită în zone, de cele două diagonale:

- diagonală principală;
- diagonală secundară.

#### Diagonală principală:

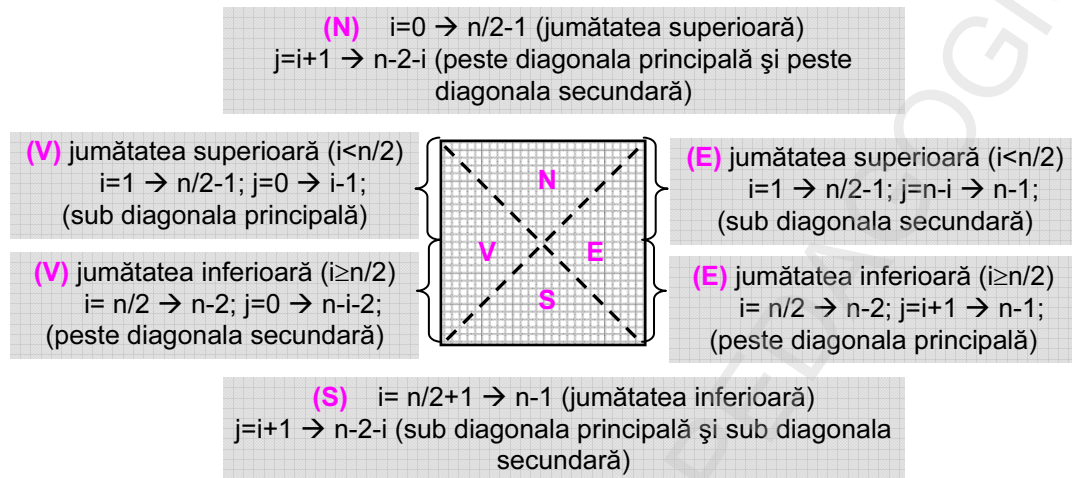


#### Diagonală secundară:





Cele două diagonale împart matricea pătrată în patru regiuni: Nord, Vest, Sud și Est:



Numărul de elemente dintr-o regiune ( $k$ ) este:

$$k = \begin{cases} (n \times n - 2 \times n) / 4 & \text{dacă } n \text{ par} \\ (n \times n - 2 \times n + 1) / 4 & \text{dacă } n \text{ impar} \end{cases}$$

**Matrice speciale** (sunt matrice care au anumite proprietăți, în funcție de valorile elementelor). Pentru a se face economie de memorie, aceste matrice pot fi memorate ca vectori.

→ **Matrice pătrată simetrică față de diagonală.** Dacă este simetrică față de diagonala principală ( $a[i][j] = a[j][i]$ ), se vor memora în vectorul  $b$  numai elementele de pe diagonala principală și cele de deasupra diagonalei principale:  $b[k] = a[i][j]$ , unde  $i=0 \rightarrow n-1$  și  $j=i \rightarrow n-1$ . Dacă este simetrică față de diagonala secundară ( $a[i][j] = a[n-i-1][n-j-1]$ ), se vor memora în vectorul  $b$  numai elementele de pe diagonala secundară și cele de deasupra diagonalei principale:  $b[k] = a[i][j]$ , unde  $i=0 \rightarrow n-1$  și  $j=0 \rightarrow n-i-1$ . Numărul de elemente ale vectorului  $b$  va fi:  $n \times (n+1) / 2$ .

→ **Matrice simetrică față de axe.** Dacă este simetrică față de axa verticală ( $a[i][j] = a[i][n-j-1]$ ), se vor memora în vectorul  $b$  numai elementele de pe axa verticală (numai dacă numărul de coloane este impar) și cele de la dreapta axei verticale:  $b[k] = a[i][j]$ , unde  $i=0 \rightarrow m-1$  și  $j=0 \rightarrow n/2$  (pentru  $n$  par) sau  $j=0 \rightarrow n/2-1$  (pentru  $n$  impar). Numărul de elemente ale vectorului  $b$  va fi:  $m \times (n/2+1)$  pentru  $n$  par, respectiv  $m \times n/2$ , pentru  $n$  impar. Dacă este simetrică față de axa orizontală ( $a[i][j] = a[m-i-1][j]$ ), se vor memora în vectorul  $b$  numai elementele de pe axa orizontală (numai dacă numărul de linii este impar) și cele de deasupra axei orizontale:  $b[k] = a[i][j]$ , unde  $i=0 \rightarrow m/2$  (pentru  $n$  par) sau  $i=0 \rightarrow m/2-1$  (pentru  $m$  impar) și  $j=0 \rightarrow n-1$ . Numărul de elemente ale vectorului  $b$  va fi:  $n \times (m/2+1)$  pentru  $m$  par, respectiv  $n \times m/2$ , pentru  $m$  impar.

→ **Matrice pătrată diagonală.** Toate elementele care nu se găsesc pe diagonala principală, respectiv secundară, au valoarea 0. Se vor memora în vectorul  $b$  numai elementele de pe diagonala principală, respectiv secundară:  $b[k] = a[i][i]$ , respectiv  $b[k] = a[i][n-i-1]$ , unde  $i=0 \rightarrow n-1$ . Numărul de elemente ale vectorului  $b$  va fi:  $n$ .

→ **Matrice pătrată triunghiulară.** Toate elementele care se găsesc deasupra sau sub o diagonală au valoarea 0. Se va memora la fel ca o matrice pătrată simetrică față de acea diagonală.

## Studiu de caz

**Scop:** exemplificarea modului în care se aplică algoritmii pentru prelucrarea matricelor pătrate.

**Enunțul problemei 1:** Să se calculeze suma elementelor de pe diagonală principală a unei matrice pătrate, de dimensiune  $n$  ( $n \leq 10$ ). Elementele matricei sunt numere întregi.

Elementele de pe diagonală principală au cei doi indici egali (numărul liniei este egal cu numărul coloanei).

```
#include <iostream.h>
void main()
{int n,i,j,s=0,a[10][10];
 cout<<"n="; cin>>n;
 for (i=0;i<n;i++) //se parcurge matricea pentru creare
 for (j=0;j<n;j++)
 {cout<<"a["<<i+1<<","<<j+1<<"]="<<"; cin>>a[i][j];}
 for (i=0;i<n;i++) s+=a[i][i]; //se parcurge diagonală principală
 cout<<"suma="<<s;}
```

**Enunțul problemei 2:** Să se verifice dacă o matrice pătrată de dimensiune  $n$  ( $n \leq 10$ ) are următoarea proprietate: elementele de pe diagonală secundară au valoarea 1, iar restul elementelor au valoarea 0. Elementele matricei sunt numere întregi.

Elementele de pe diagonală secundară au suma indicilor egală cu  $n-1$  ( $i+j=n-1$ ). Se va folosi o variabilă  $x$ , care va avea valoarea 1 dacă matricea are proprietatea specificată (valoarea logică True), și valoarea 0 dacă matricea nu are proprietatea specificată (valoarea logică False). Variabila  $x$  se inițializează cu 1 (presupunem că matricea are proprietatea specificată). Deoarece, pentru a determina proprietatea matricei, nu trebuie parcurse toate elementele, în cazul în care se găsește un element care nu îndeplinește condiția din proprietate, parcurgerea matricei se va face cu o instrucțiune **while**. Se vor parcurge coloanele unei linii prin incrementarea indicelui  $j$ , după care se va trece la linia următoare, prin incrementarea indicelui  $i$  și inițializarea cu 0 a indicelui  $j$ .

```
#include <iostream.h>
void main()
{int n,i,j,x=1,a[10][10];
 cout<<"n="; cin>>n;
 for (i=0;i<n;i++)
 for (j=0;j<n;j++)
 {cout<<"a["<<i+1<<","<<j+1<<"]="<<"; cin>>a[i][j];}
 i=0; j=0;
 while (i<n && x)
 {if (i+j==n-1)
 {if (a[i][j]!=1) x=0;}
 else
 if (a[i][j]!=0) x=0;
 if (j==n-1){j=0; i++;}
 else j++;}
 if (x) cout<<"Matricea are proprietatile specificate";
 else cout<<"Matricea nu are proprietatile specificate"; }
```




**Temă**

Următoarele probleme se vor descompune în subprobleme și fiecare subproblemă va fi implementată cu ajutorul unui **subprogram**.

1. Se citește de la tastatură un număr natural  $n$  care reprezintă dimensiunea unei matrice pătrate cu numere întregi. Elementele matricei se citesc de la tastatură. Afișați suma elementelor de pe cele două diagonale.
2. Să se verifice dacă o matrice pătrată cu dimensiunea  $n \times n$  este:
  - a. simetrică față de axa orizontală,
  - b. simetrică față de axa verticală,
  - c. simetrică față de diagonala principală,
  - d. simetrică față de diagonala secundară.
3. Se consideră o matrice pătrată cu dimensiunea  $n \times n$  și un vector cu  $n$  elemente. Numărul  $n$  și elementele matricei și ale vectorului se citesc de la tastatură. Să se verifice dacă elementele vectorului formează o linie sau o coloană a matricei. În caz afirmativ, să se afișeze un mesaj în care să se precizeze numărul liniei și/sau al coloanei.
4. Să se genereze toate matricele binare pătrate de dimensiune  $n$  care au un singur element de 1 pe linie și un singur element de 1 pe coloană. O matrice binară este o matrice ale cărei elemente au valoarea 0 sau 1. **Indicație.** Soluția are  $n$  elemente. Elementul soluției  $x_k$  reprezintă numărul coloanei de pe linia  $k$  pe care se găsește elementul cu valoarea 1.
5. Se consideră o matrice pătrată  $a$  cu numere întregi, cu dimensiunea  $n$ . Folosind metoda **divide et impera**:
  - a. să se interschimbe linia  $p$  cu coloana  $q$  ( $p$  și  $q$  se citesc de la tastatură);
  - b. să se interschimbe diagonala principală cu diagonala secundară.

**Evaluate**
**Răspundeți:**

1. Se consideră o matrice cu 3 linii și 5 coloane. Arătați cum vor fi aranjate în memorie elementele matricei dacă ele sunt înregistrate în ordinea liniilor. Arătați cum vor fi aranjate și în cazul în care ar fi înregistrate în ordinea coloanelor. Ce constatați? Cum vor fi aranjate în memorie elementele acestei matrice dacă va fi implementată în limbajul C++?
2. Construiți o structură de date de tip tabel, în care să înregistrați notele elevilor din clasă pe semestrul 1, la disciplina „Informatică”.
3. Dacă elementele unei matrice cu 5 linii și 8 coloane sunt înregistrate în ordinea liniilor, începând de la adresa de memorie 100, care este adresa elementului din linia 4 și coloana 5? Datele memorate în matrice sunt de tip **int**.
4. Dacă elementele unei matrice cu  $m$  linii și  $n$  coloane sunt înregistrate în memoria internă în ordinea coloanelor, care este formula prin care calculați numărul de ordine al elementului din linia  $i$  și coloana  $j$ ?
5. Cum se declară un vector cu 10 elemente de tip întreg care să aibă valoarea inițială 0?
6. Găsiți greșelile din următoarele instrucțiuni declarative:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 5 | 4 | 3 | 2 | 1 |
| 1 | 3 | 5 | 2 | 4 |

```
const int DIM=10;
int n, a[n][n], b[dim][dim], B[10][10], c[10,20];
```

7. Care este succesorul elementului  $a[i][j]$  al matricei definită prin instrucțiunea declarativă `int a[4][5];`? Dar predecesorul său?
8. Fie matricea cu 3 linii și 5 coloane de tip `int` în care se înregistrează, coloană cu coloană, primele 15 numere naturale pare ( $a[0][0]=2$ ;  $a[1][0]=4$ ; ...,  $a[0][1]=8$ ; etc.). Elementele matricei **a** se copiază, în ordine, linie cu linie, în vectorul **b** care are 15 elemente de tip `int`. Ce valoare are elementul  $b[5]$ ?

### Adevărat sau Fals:

1. Tabloul de memorie este o structură de date externă.
2. Tabloul de memorie se creează într-o zonă continuă de memorie internă.
3. Tabloul de memorie este întotdeauna o structură de date temporară.
4. Tabloul de memorie de tip matrice nu este o structură de date liniară.
5. Pentru indicii unei matrice se pot folosi numere reale.
6. Cu declarațiile următoare, se poate folosi variabila **x**, pentru a memora 15 numere întregi:

```
typedef int vec int[3];
vec_int x[5];
```

7. Pentru variabila **x** definită anterior, este corectă atribuirea:  $x[2] = \{1, 2, 3\}$ ;
8. Cu declarațiile următoare, se obțin tablourile de memorie **x** și **y**, echivalente din punct de vedere al implementării:

```
typedef int vec int[3];
vec_int x[5];
int y[5][3];
```

9. Cu declarația `int x[3][5], y[5][3];` se obțin tablourile de memorie **x** și **y**, echivalente din punct de vedere al implementării:
10. Cu declarația `int x[15], y[5][3];` se obțin tablourile de memorie **x** și **y**, echivalente din punct de vedere al implementării:
11. În matricea pătrată **a**, cu **n** linii și **n** coloane, elementul  $a[n-2][n-2]$  aparține diagonalei principale.
12. În matricea pătrată **a**, cu **n** linii și **n** coloane, elementul  $a[n-2][3]$  aparține diagonalei secundare.
13. În matricea pătrată **a**, cu 10 linii și 10 coloane, elementul  $a[3][2]$  se găsește deasupra diagonalei principale.
14. În matricea pătrată **a**, cu 10 linii și 10 coloane, elementul  $a[3][2]$  se găsește sub diagonala secundară.
15. Secvența următoare de program calculează minimul elementelor de pe diagonala principală a unei matrice pătrate **a**, de dimensiune  $n \times n$ :

```
int n, i, x, a[50][50];
```

```
.....
for (i=1; i<n; i++)
 if (a[i][i]<a[0][0])
 {x=a[i][i]; a[i][i]=a[0][0]; a[0][0]=x;}
```

### Alegeți:

1. Se consideră următoarea matrice, stocată în memoria internă a calculatorului în ordinea coloanelor. Elementul cu numărul de ordine 5 este:

|   |   |   |   |   |
|---|---|---|---|---|
| a | b | c | d | e |
| f | g | h | i | j |
| k | l | m | n | o |

- a) e
- b) g
- c) h

2. Se consideră următoarea matrice, cu numele *alfa*, stocată în memoria internă a calculatorului în ordinea liniilor. Elementul *alfa*(2,3) are valoarea:

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 |

- a) 12  
b) 8  
c) 6

3. Care dintre următoarele secvențe de instrucțiuni determină, în variabila întreagă *s*, suma elementelor de sub diagonala secundară a unei matrice pătrate *a*, cu numere întregi, cu dimensiunea  $n \times n$  ?

a) *s*=0;

```
for (i=0;i<n;i++)
 for (j=i+1;j<n;i++)
 s+=a[i][j];
```

b) *s*=0;

```
for (i=0;i<n;i++)
 for (j=i;j<n;i++)
 s+=a[i][j];
```

c) *s*=0;

```
for (i=0;i<n;i++)
 for (j=0;j<=i-1;i++)
 s+=a[i][n-j-1];
```

d) *s*=0;

```
for (i=1;i<n;i++)
 for (j=1;j<=i;i++)
 s+=a[i][n-j];
```

4. Secvența următoare de program realizează:

```
int n,i,x,p,q,a[20][20];
.....
for (i=0;i<n;i++)
 {x=a[i][q]; a[i][q]=a[i][p]; a[i][p]=x}
```

- a. sortarea elementelor de pe linia *i* folosind metoda selectării directe;  
b. interschimbarea coloanei *p* cu coloana *q*;  
c. interschimbarea liniei *p* cu linia *q*;  
d. sortarea elementelor de pe linia *i* folosind metoda bulelor.

## Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra în **echipă**. Fiecare miniproiect va conține:

- descompunerea problemei în subprobleme și rezolvarea fiecărei subprobleme cu ajutorul unui subprogram.**
- explicarea algoritmilor folosiți pentru realizarea subprogramelor;
- citirea matricelor dintr-un fișier text;
- alte două exemple de probleme cu aplicare practică, în care, pentru implementarea datelor, se vor folosi tablouri bidimensionale, și rezolvarea acestor probleme. Exemplu. *Se înregistrează într-un tabel cheltuielile lunare cu utilitățile (electricitate, apă, gaze, telefon, cablu TV etc.) pe perioada unui an. Trebuie să se determine: media consumului anual și trimestrial, pe fiecare tip de cheltuială și pentru cheltuielile totale, luna cu cele mai multe cheltuieli și luna cu cele mai puține cheltuieli.*

1. Găsiți metoda adecvată prin care să memorați coordonatele carteziene în spațiu ( $x_i$ ,  $y_i$ ,  $z_i$ ) a *n* vectori, astfel încât să puteți implementa un algoritm cât mai eficient care să afișeze vectorii perpendiculari (vectorii al căror produs scalar este 0). Afișați vectorii perpendiculari.

Pentru următoarele probleme, se consideră o matrice pătrată cu elemente numere întregi cu dimensiunea  $n \times n$ .

2. Să se afișeze:
  - a. suma elementelor situate deasupra diagonalei principale;
  - b. simetrica matricei față de diagonala secundară;
  - c. descrescător, elementele de pe diagonala principală, folosind metoda selecției directe.
3. Să se afișeze:
  - a. suma elementelor situate sub diagonala secundară;
  - b. simetrica matricei față de diagonala principală;
  - c. crescător, elementele de pe diagonala secundară, folosind metoda bulelor.
4. Să se afișeze:
  - a. suma elementelor situate sub diagonala principală;
  - b. simetrica matricei față de axa orizontală care trece prin centrul matricei;
  - c. crescător, elementele de pe linia **p**, folosind metoda sortării prin interclasare (**p** se citește de la tastatură).
5. Să se afișeze:
  - a. suma elementelor situate deasupra diagonalei secundare;
  - b. simetrica matricei față de axa verticală care trece prin centrul matricei;
  - c. descrescător, elementele de pe coloana **q**, folosind metoda sortării rapide (**q** se citește de la tastatură).
6. Să se afișeze:
  - a. elementele situate deasupra diagonalei principale;
  - b. suma elementelor pare din regiunea Vest a matricei;
  - c. descrescător, elementele de pe diagonala secundară, folosind metoda bulelor.
7. Să se afișeze:
  - a. elementele situate sub diagonala secundară;
  - b. media aritmetică a elementelor din regiunea Nord a matricei;
  - c. crescător, elementele de pe linia **p**, folosind metoda sortării rapide (**p** se citește de la tastatură).
8. Să se afișeze:
  - a. elementele situate sub diagonala principală;
  - b. produsul elementelor impare din regiunea Est a matricei;
  - c. crescător, elementele de pe diagonala secundară, folosind metoda selecției directe.
9. Să se afișeze:
  - a. elementele situate deasupra diagonalei secundare;
  - b. numărul de elemente, care au ultima cifră divizibilă cu 3, din regiunea Sud a matricei;
  - c. descrescător, elementele de pe coloana **q**, folosind metoda sortării prin interclasare (**q** se citește de la tastatură).

## 2.3. Șirul de caractere

Un cuvânt, o propoziție, o frază, un paragraf, un text – reprezintă o **mulțime ordonată de caractere** care poate avea o lungime variabilă.

**Șirul de caractere** este o structură de date care este formată dintr-o mulțime ordonată de caractere, în care fiecare caracter se identifică prin poziția sa în cadrul mulțimii.

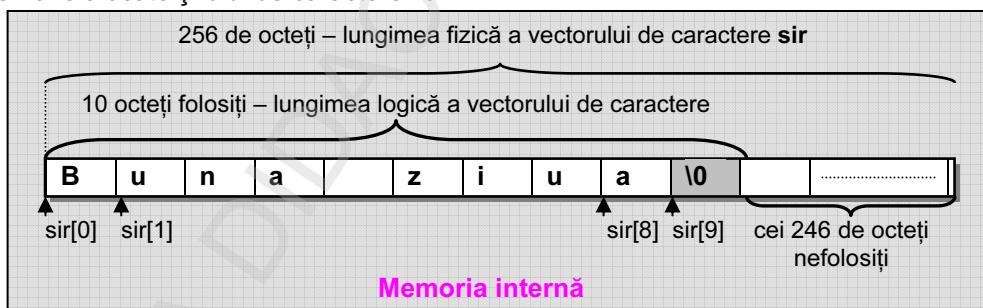
### 2.3.1. Implementarea șirului de caractere în C++

În limbajul C++, implementarea șirurilor de caractere se face sub forma unui tablou unidimensional (vector) ale cărui elemente sunt de tip caracter, fiecare caracter fiind reprezentat prin codul său ASCII.

Ați aflat că, în general, vectorii au două lungimi: o lungime fizică (care reprezintă numărul maxim de elemente pe care le poate avea vectorul și corespunde locațiilor de memorie rezervate în urma declarării lui) și o lungime logică (numărul de elemente folosite din vector, la o execuție a programului). Și în cazul unui vector de caractere există o lungime fizică și o lungime logică a vectorului. Ceea ce deosebește un vector de caractere de vectorii cu alte tipuri de elemente este posibilitatea de a marca sfârșitul logic al vectorului prin folosirea **caracterului NULL** – specificat prin constanta caracter **'\0'** (care are **codul ASCII 0**). Acest caracter se adaugă la sfârșitul caracterelor memorate în vector, ca un **terminator**. De exemplu, prin instrucțiunea:

```
char sir[256];
```

se creează un șir de caractere în care vor putea fi memorate maxim 255 de caractere, deoarece un element al vectorului se va folosi pentru a memora caracterul NULL. Astfel, dacă în acest vector se memorează șirul de caractere "Buna ziua", conținutul zonei de memorie alocate șirului de caractere va fi:



Într-un șir de caractere, ordinea acestora este esențială, fiecărui caracter putând să i se asocieze un număr care reprezintă poziția caracterului în cadrul șirului, iar fiecare caracter din șir poate fi identificat, ca și în cadrul unui vector, prin poziția sa în cadrul șirului: **sir[i]**. Astfel, în șirul declarat anterior, **sir[3]** reprezintă caracterul din poziția 4, numărarea pozițiilor începând de la 0, adică al patrulea caracter al șirului: **a**.

#### Atenție

Vectorul de caractere trebuie declarat cu un caracter mai mult (caracterul **'\0'**) decât cel mai mare șir de caractere pe care îl poate conține. De exemplu, o mulțime ordonată de maxim 20 de caractere va putea fi descrisă astfel: **char sir[21];**.



În exemplul anterior a fost declarat un șir de caractere precizând lungimea sa maximă. Puteți să inițializați un șir de caractere la declararea lui, atribuindu-i o constantă de tip șir de caractere. O **constantă de tip șir de caractere** este o succesiune de caractere delimitată de **ghilimele**.

```
char sir[256]="Buna ziua";
```

Prin această instrucțiune s-a declarat un șir de caractere cu lungimea maximă de 256 de caractere (pentru care s-au rezervat 256 de octeți) din care au fost ocupate numai 10 caractere: nouă pentru șirul de caractere "Buna ziua" și unul pentru **caracterul NULL care este adăugat automat de către compilator**. Dacă se inițializează la declarare un șir de caractere, pozițiile neocupate din șir vor fi completate cu caracterul NULL.

**Exemplu:**

```
char sir[20]="Buna ziua"; int i;
for (i=0;i<20;i++)
 if (sir[i]==NULL) cout<<'0'; else cout<<sir[i];
/* se afișează Buna ziua000000000000 */
```

Dacă se inițializează șirul de caractere, nu mai este obligatoriu să se precizeze lungimea maximă a șirului, aceasta fiind calculată de către compilator. De exemplu, prin instrucțiunea:

```
char sir[]="Buna ziua";
```

s-a declarat un șir de caractere pentru care compilatorul va rezerva numărul de octeți necesari pentru memorarea constantei șir de caractere (9 octeți) și a caracterului NULL (1 octet), adică 10 octeți.

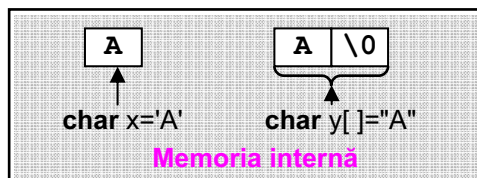
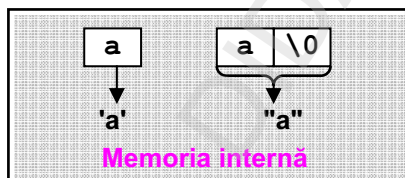
### Atenție

După ce ați declarat un șir de caractere nu mai puteți să-i atribuiți o constantă de tip șir de caractere. Numele șirului de caractere este numele unui tablou de memorie, deci o constantă de tip adresă:

```
char sir[20]; sir="Buna ziua";
/* Eroare! valoarea unei constante nu poate fi modificată */
```

### Atenție

O constantă de tip șir de caractere, chiar dacă va conține un singur caracter, este diferită de o constantă de tip caracter, datorită modului diferit de stocare în memoria internă. În același mod, un șir de caractere, chiar dacă va conține un singur caracter, este diferit de o dată elementară de tip **char**:



Pentru a insera caracterul ghilimele într-o constantă de tip șir de caractere, folosiți secvența escape **"\"**. Astfel, pentru a afișa pe ecran textul **Acesta este un "Exemplu"** folosiți instrucțiunea: `cout<<"Acesta este un \"Exemplu\"";`.

**Lungimea unui șir de caractere** reprezintă numărul de caractere din șir, mai puțin caracterul NULL.

De exemplu, lungimea șirului de caractere declarat prin instrucțiunea următoare este 9, chiar dacă se vor rezerva 10 octeți pentru memorarea lui:

```
char sir[]="Buna ziua";
```

**Șirul vid** sau **șirul nul** este șirul care are lungimea 0.

De exemplu, puteți să inițializați un șir de caractere la declararea lui ca șir nul, astfel:

```
char sir[256]="";
```

În cadrul programului, puteți să inițializați un șir de caractere ca șir nul atribuind primei poziții din șir valoarea NULL printr-una dintre următoarele instrucțiuni de atribuire:

```
sir[0]=NULL; ↔ sir[0]=0; ↔ sir[0]='\0';
```

### Atenție

Operațiile de atribuire `sir[0]="";`, `sir[0]='';` sau `sir[0]='0';` nu pot fi folosite pentru inițializarea unui șir vid. Primele două vor produce eroare la compilare, deoarece nu se poate atribui unui element care este de tip **char** o constantă de tip șir de caractere, respectiv nu există un caracter ASCII care să fie precizat prin constanta ". În cel de al treilea caz, primul caracter din șir va fi caracterul cifra 0, iar lungimea șirului va fi determinată de existența unui caracter NULL în cadrul șirului.

### Temă



Declarați trei șiruri de caractere pe care le inițializați astfel încât să aibă lungimea 10, 1 și respectiv 0.

## 2.3.2. Citirea și scrierea șirurilor de caractere

Citirea și scrierea șirurilor de caractere se poate face:

- la nivel de element al structurii – caracterul;
- la nivel de structură – șirul de caractere.

**Exemplul 1** – Se citește și se afișează un șir de caractere. Citirea se face caracter cu caracter, până la apăsarea tastei **Enter** (reprezentarea codului ASCII al caracterului **Enter** se face prin secvența escape '\n'). Afișarea se face caracter cu caracter, până la întâlnirea caracterului NULL care marchează sfârșitul șirului de caractere. Citirea se face cu format, folosind manipulatorul care permite citirea caracterelor albe.

```
#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
void main()
{char sir[256]; int i=0;
 cin>>resetiosflags(ios::skipws)>>sir[i];
 while (sir[i]!='\n') {i++; cin>>resetiosflags(ios::skipws)>>sir[i];}
 sir[i+1]=0; //se adaugă caracterul NULL
 for (i=0;sir[i]!=NULL;i++) cout<<sir[i];}
```

### Observație:

În cazul în care citirea unui șir de caractere se face caracter cu caracter, trebuie adăugat caracterul NULL la sfârșitul șirului de caractere.

**Exemplul 2** – Se scriu într-un șir de caractere literele alfabetului latin: aAbBcC...zZ și apoi se afișează literă cu literă.

```
#include <iostream.h>
#include <stdlib.h>
void main()
{char sir[256]; int i,j;
```

```
for (i=0,j=0;j<26;j++) {sir[i]='a'+j; sir[i+1]='A'+j; i+=2;}
sir[i+1]=NULL; //se adaugă caracterul NULL
for (i=0;sir[i]!=NULL;i++) cout<<sir[i];}
```

### Temă



1. Scrieți secvența de instrucțiuni prin care atribuiți unui șir de caractere valoarea "0123...89abc...zABC...Z98...3210" și afișați apoi acest șir, caracter cu caracter.

2. Să se afișeze toate anagramele unui cuvânt citit de la tastatură.

**Exemplul 3** – Se citește și se afișează un șir de caractere, operațiile executându-se la nivelul structurii de date.

```
#include <iostream.h>
void main() {char sir[256]; cin>>sir; cout<<sir;}
```

#### Observații:

1. Acest mod de citire și scriere a șirurilor de caractere este mult mai simplu, deoarece caracterul NULL este adăugat automat de către compilator.
2. Primul caracter scris în șirul de caractere va fi primul caracter citit de la tastatură care nu este un caracter alb. Operația de citire de la tastatură se va termina la întâlnirea unui caracter alb (de exemplu, spațiu sau apăsarea tastei **Enter**). Dezavantajul acestei metode îl reprezintă faptul că nu pot fi citite de la tastatură caracterele albe, cum este de exemplu spațiu (nu poate fi citit un text care conține mai multe cuvinte). Astfel, dacă se introduce de la tastatură în variabila de memorie *sir* șirul de caractere `↵↵↵alfa↵↵beta` (caracterul `↵` simbolizează un spațiu), pe ecran se va afișa *alfa*.

Pentru a elimina acest dezavantaj, se poate folosi funcția **get()** cu următoarele forme, care diferă prin parametri folosiți:

**Forma 1** (cu parametri):

```
cin.get(sir,nr,ch);
```

Parametrii funcției sunt: **sir** de tip șir de caractere, **nr** de tip întreg și **ch** de tip caracter, ultimul fiind opțional. Efectul acestei funcții este următorul: se citesc de la tastatură mai multe caractere, inclusiv caracterele albe, care vor fi scrise în variabila **sir**, până când se produce unul dintre următoarele evenimente:

- au fost citite maxim **nr-1** caractere;
- a fost întâlnit caracterul **ch** care are rolul de delimitator – el nu va fi scris în variabila **sir** și nu va fi înlăturat din fluxul de intrare; dacă nu este precizat acest parametru, se consideră implicit caracterul `'\n'` (linie nouă) generat la apăsarea tastei **Enter**.

#### Exemplu:

```
char sir[256]; cin.get(sir,5); cout<<sir;
```

Se scriu în variabila **sir** maxim 4 caractere introduse de la tastatură. De exemplu, dacă introduceți de la tastatură *Calculator* **Enter**, se va afișa *Calc*, iar dacă introduceți *An* **Enter**, se va afișa *An*.

#### Exemplu:

```
char sir[256]; cin.get(sir,9,'#'); cout<<sir;
```

Se scriu în variabila **sir** caracterele introduse până la întâlnirea caracterului **#**, dar nu mai mult de 8 caractere. De exemplu, dacă introduceți de la tastatură *alfa#beta* **Enter**, se va afișa *alfa*, iar dacă introduceți *alfa beta* **Enter**, se va afișa *alfa bet*.

**Forma 2** (fără parametri):

```
cin.get();
```

Această funcție nu are nici un parametru și furnizează următorul caracter din fluxul de date sub forma unei valori întregi. Ea este folosită, după o funcție `cin.get()` cu parametri, pentru a descărca din fluxul de date ultimul caracter citit (delimitatorul), care ar împiedica efectuarea unei a doua operații de citire de la tastatură, dacă funcția `cin.get()`, cu care se face a doua citire, folosește același caracter ca delimitator. De obicei, într-un program, în fluxurile de date de la tastatură, se folosește ca delimitator pentru operațiile de citire caracterul linie nouă, generat prin apăsarea tastei **Enter**. Dacă acest caracter rămâne în flux după prima operație de citire, iar a doua operație de citire folosește același delimitator, primul caracter care va fi interpretat de cea de a doua operație de citire va fi caracterul linie nouă care, fiind delimitator, va determina terminarea operației de citire de la tastatură.

**Exemplu:**

```
char sir1[256], sir2[256];
cin.get(sir1,5); cin.get(); cin.get(sir2,5);
cout<<sir1<<sir2;
```

De exemplu, dacă introduceți de la tastatură *alfa beta gama delta* **Enter**, se va afișa *alfabeta*, *alfa* fiind valoarea variabilei `sir1`, iar *beta*, valoarea variabilei `sir2`. Dar, dacă introduceți *a1* **Enter** *a2* **Enter**, se va afișa *a1a2*, *a1* fiind valoarea variabilei `sir1`, iar *a2*, valoarea variabilei `sir2`. Dacă eliminați din secvență instrucțiunea `cin.get();` și introduceți aceleași date, în primul caz se va afișa *alfa bet*, *alfa* fiind valoarea variabilei `sir1`, iar *bet*, valoarea variabilei `sir2`, iar în al doilea caz, după ce veți introduce de la tastatură *a1* **Enter** se va afișa *a1*, *a1* fiind valoarea variabilei `sir1`, iar variabila `sir2` nu va conține nimic, deoarece primul caracter citit pentru această variabilă va fi caracterul linie nouă, care va determina terminarea operației de citire.

### Temă



Considerând următoarea secvență de instrucțiuni, precizați ce se va afișa dacă se va citi de la tastatură *alfa beta gama* **Enter**? Dar dacă se citește *alfa#beta gama* **Enter**? Dar dacă se citește *alfa* **Enter** *beta***Enter**? Ce se va întâmpla dacă din secvență se va elimina instrucțiunea `cin.get();` și veți relua cele trei variante de introducere a datelor?

```
char sir1[256], sir2[256];
cin.get(sir1,5); cin.get(); cin.get(sir2,5,'#');
cout<<sir1<<sir2;
```

## Șirurile de caractere și pointerii

Șirurile de caractere pot fi manipulate prin intermediul unei variabile de tip pointer către tipul `char`.

```
char *p,*q;
p="Ana are mere."; cout<<p; //afișează Ana are mere.
q=p+2; cout<<q; //afișează a are mere.
cout<<q-p; //afișează 2
```

Din această cauză, un șir de caractere poate fi declarat ca un **pointer către tipul char**, cum este de exemplu declarația șirului de caractere `sir` din exemplul următor:

```
char *sir="ala bala portocala",*p;
cout<<strlen(sir)<<" "<<sir<<endl; //afișează 18 ala bala portocala
sir++; cout<<strlen(sir)<<" "<<sir<<endl;
//afișează 17 la bala portocala
p=sir; p+=3; cout<<strlen(p)<<" "<<p; //afișează 14 bala portocala
```

De exemplu, declararea unui șir de caractere cu `char *sir="abcd"` poate fi folosită în locul declarației `char sir[]="abcd"`.

Cele două declarații au următoarele **caracteristici comune**:

- În ambele cazuri, **sir** reprezintă o adresă și, pe acest identificator, se pot aplica următorii operatori folosiți pentru tipul de dată pointer:
  - operatorul de indirectare **\*** (**\*sir**) care furnizează conținutul variabilei de la adresa indicată: în primul caz, caracterul memorat în octetul de la adresa indicată de pointerul **sir**, iar în al doilea caz caracterul memorat în primul element al vectorului;
  - operatorul aritmetic pentru adunarea unei constante **+** (**sir+1**) care furnizează o adresă: în primul caz, caracterul memorat în octetul care urmează celui indicat de pointerul **sir**, iar în al doilea caz caracterul memorat în al doilea element al vectorului;
  - operatorul aritmetic pentru scăderea a doi pointeri **-** (**sir1-sir2**) care furnizează în ambele exemple numărul de elemente dintre cele două adrese de memorie;
  - operatorul indice **[]**: **sir[i] ↔ \*(sir+i)**.
- În ambele cazuri, se poate folosi instrucțiunea **cout<<sir;** pentru a afișa conținutul șirului de caractere. Această instrucțiune afișează ceea ce este stocat de la adresa memorată în pointerul **sir**, respectiv de la adresa simbolică **sir**, până la întâlnirea caracterului **NULL**.

**Deosebiri**le dintre cele două declarații sunt:

|                                           | <b>char sir[]="abcd"</b>                                                                                                                                                                            | <b>char *sir="abcd"</b>                                                                                                                                                                  |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>semnificația identificatorului sir</b> | Este numele simbolic al unei <b>constante de tip adresă</b> : valoarea sa nu se poate modifica.                                                                                                     | Este numele unui pointer către tipul <b>char</b> , adică numele unei <b>variabile de tip adresă</b>                                                                                      |
| <b>alocarea memoriei interne</b>          | Se <b>alocă efectiv 5 octeți</b> , începând de la adresa simbolică <b>sir</b> , pentru cele 5 componente ale vectorului de caractere (șirul de caractere).                                          | Compilerul alocă 5 octeți pentru constanta de tip șir de caractere "abcd" și creează o variabilă de memorie de tip pointer care conține adresa primului caracter din șir.                |
| <b>operatori</b>                          | Deoarece <b>sir</b> este o constantă, nu se pot aplica pe identificatorul <b>sir</b> operatorii de incrementare și decrementare și nici nu i se poate modifica valoarea prin operația de atribuire. | Deoarece <b>sir</b> este o variabilă, se pot aplica pe identificatorul <b>sir</b> operatorii de incrementare și decrementare și i se poate modifica valoarea prin operația de atribuire. |
| <b>citirea de la tastatură</b>            | <b>cin.get(sir,100)</b> – citește de la tastatură un șir de maxim 100 de caractere care va fi scris în memorie începând de la adresa simbolică <b>sir</b> .                                         | <b>cin.get(*sir)</b> – citește de la tastatură un caracter care va fi scris la adresa memorată în pointerul <b>sir</b> .                                                                 |

### 2.3.3. Algoritmi pentru prelucrarea șirurilor de caractere

Prelucrarea șirurilor de caractere se poate face prin mai multe metode:

- prin **parcurgerea caracterelor din șir** ca elemente ale unui vector de caractere:
  - folosind indicii sau
  - folosind pointerii;
- folosind funcțiile de sistem** implementate în bibliotecile limbajului; fișierul antet în care sunt definite cele mai multe dintre aceste funcții este fișierul **<string.h>**.

**Exemplu – Se afișează lungimea unui șir de caractere.**

```
#include <iostream.h>
#include <stdlib.h>
void main()
{char sir[256]; int i;
 cout<<"Introduceți sirul de caractere"<<endl; cin.get(sir,100);
 for (i=0;sir[i]!=NULL;i++); //sau for(i=0;sir[i];i++);
 cout<<"Lungimea sirului de caractere este "<<i;}
```

Aceeași problemă se poate rezolva ținând cont că parcurgerea unui vector se poate face și cu ajutorul pointerilor. Folosind un pointer **p** către elementele șirului, se parcurge șirul de la prima poziție până la sfârșitul lui, prin aplicarea operatorului de incrementare pe pointer. Lungimea șirului se obține făcând diferența dintre adresa memorată în pointerul **p** și adresa primului element din șir.

```
#include <iostream.h>
void main()
{char sir[256],*p;
 cout<<"Introduceți sirul de caractere"<<endl; cin.get(sir,100);
 for (p=sir;*p!=0;p++); //sau for (p=sir;*p;p++);
 cout<<"Lungimea sirului de caractere este "<<p-sir;}
```

Rezolvarea acestei probleme este și mai simplă dacă se folosește funcția sistem **strlen()** care are sintaxa: **strlen(sir)**:

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256];
 cout<<"Introduceți sirul de caractere"<<endl; cin.get(sir,100);
 cout<<"Lungimea sirului de caractere este "<<strlen(sir);}
```

## Șirurile de caractere și subprogramele

### a. Subprogramele utilizator

Transmiterea unui parametru de tip **șir de caractere** se poate face fie folosind pointeri, fie folosind vectorii de caractere. De exemplu, următoarele două anteturi de funcții transmit ca parametri două șiruri de caractere, și sunt echivalente:

- a. **void copiaza(char sir1[], char sir2[])**
- b. **void copiaza(char \*sir1, char \*sir2)**

Următoarele patru exemple de funcții realizează aceeași operație: copiază un șir de caractere în alt șir de caractere:

|                                                                                                                                                                             |                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>char *copiaza(char *d, char *s) {char *p=d;  while (*s) *d++=*s++;  *d=*s;  return p;} void main() {char a[256]="alfabet",b[256];  copiaza(b,a); cout&lt;&lt;b;}</pre> | <pre>void copiaza(char *d, char *s) {while (*s) *d++=*s++;  *d=*s;} void main() {char a[256]="alfabet",b[256];  copiaza(b,a); cout&lt;&lt;b;}</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|

```
char *copiază(char d[], char s[])
{char *p=d;
for (int i=0; s[i]!=NULL;i++)
 d[i]=s[i];
di]=s[i];
return p;}
void main()
{char a[256]="alfabet",b[256];
copiază(b,a); cout<<b;}
```

```
void copiază(char d[], char s[])
{for (int i=0; s[i]!=NULL;i++)
 d[i]=s[i];}
void main()
{char a[256]="alfabet",b[256];
copiază(b,a); cout<<b;}
```

**Observație.** Spre deosebire de vectorii numerici, în cazul șirurilor de caractere nu trebuie să precizați lungimea logică, aceasta fiind determinată de poziția caracterului NULL.



**Temă**

Scrieți un subprogram care să testeze dacă un text citit de la tastatură este un cuvânt (condiția ca să fie cuvânt este să conțină numai litere).

### a. Subprogramele de sistem

Pentru a putea folosi o funcție de sistem trebuie să știți să interpretați prototipul funcției pe care vi-l pune la dispoziție autodocumentarea (**help**-ul) limbajului de programare.

**Exemplul 1** – Funcția **strcpy()** din fișierul antet **string.h** copiază șirul de caractere sursă **src** în șirul de caractere destinație **dest**, precizate prin parametri. Ea are prototipul:

```
char *strcpy(char *dest, const char *src);
```

Interpretați prototipul funcției astfel:

- Rezultatul funcției este de tip adresa unui șir de caractere – pointer către tipul **char**.
- Funcția are doi parametri de tip șir de caractere (precizați prin pointeri către tipul **char**).
- Cuvântul cheie **const** care precede cel de al doilea parametru precizează faptul că subprogramul nu trebuie să modifice cel de al doilea șir de caractere. Dacă funcția ar încerca să modifice acest parametru, compilatorul va semnala o eroare.

**Exemplul 2** – Funcția **ecvt()** din fișierul antet **stdlib.h** furnizează, prin numele ei, un șir de caractere în care a fost convertit un număr real **value**. Șirul de caractere are lungimea **ndig**. Poziția punctului zecimal este **dec**, iar semnul este **sign**. Ea are prototipul:

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Interpretați prototipul funcției, astfel:

- Rezultatul funcției este de tip șir de caractere – pointer către tipul **char**.
- Funcția are patru parametri: unul de tip real pentru **value** – **double**, unul de tip întreg pentru lungimea șirului de caractere **ndig** – **int**, unul de tip adresă către întreg pentru poziția punctului zecimal **ndig** – **\*int** și unul de tip adresă către întreg pentru semnul numărului **sign** – **\*int**.
- Parametrii **dec** și **sign** sunt parametri de ieșire. Ei se transmit prin valoare, folosind adrese de memorie (pointeri către întreg).

**Algoritmii pentru prelucrarea șirurilor de caractere** pot fi grupați astfel:

1. prelucrarea a două șiruri de caractere;
2. prelucrarea unui șir de caractere;
3. prelucrarea subșirurilor de caractere;
4. conversii între tipul șir de caractere și tipuri numerice.



### 2.3.3.1. Prelucrarea a două șiruri de caractere

Pentru prelucrarea a două șiruri de caractere puteți folosi următoarele operații:

- copierea unui șir de caractere într-un alt șir de caractere;
- concatenarea a două șiruri de caractere;
- compararea a două șiruri de caractere.

#### Copierea unui șir de caractere într-un alt șir de caractere

Se transferă conținutul unui șir de caractere (**șirul sursă**) într-un alt șir de caractere (**șirul destinație**). Puteți să folosiți următoarele funcții de sistem (parametrii funcțiilor sunt: **s1** – șirul sursă, **s2** – șirul destinație și **n** numărul de caractere care se copiază):

| Funcția          | Sintaxa apelului          | Realizează                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strcpy()</b>  | <b>strcpy(s2, s1)</b>     | Sunt copiate din șirul sursă <b>s1</b> în șirul destinație <b>s2</b> toate caracterele, inclusiv caracterul NULL. Funcția furnizează ca rezultat un pointer care indică adresa șirului destinație.                                                                                                                                                                                                                      |
| <b>strncpy()</b> | <b>strncpy(s2, s1, n)</b> | Sunt copiate din șirul sursă <b>s1</b> în șirul destinație <b>s2</b> maxim <b>n</b> caractere, începând cu primul caracter. Dacă lungimea șirului sursă este mai mică decât <b>n</b> , va fi copiat și caracterul NULL – funcția fiind echivalentă cu <b>strcpy()</b> ; altfel, șirul destinație nu va fi terminat cu caracterul NULL. Funcția furnizează ca rezultat un pointer care indică adresa șirului destinație. |

Exemplu:

```
#include <iostream.h>
#include <string.h>
void main()
{char sir1[256], sir2[256];
 cout<<"Sirul de caractere care se copiază"<<endl; cin.get(sir1,100);
 strcpy(sir2,sir1); cout<<sir2;}
```

#### Atenție

Dacă vreți să atribuiți unei variabile de tip șir de caractere **sir** o constantă de tip șir de caractere **const\_sir**, folosiți funcția **strcpy()** astfel: **strcpy(sir, const\_sir)**. Următoarele secvențe de program sunt echivalente:

```
char sir[256]="alfa";
```

**secvența 1**

```
char sir[256];
```

```
.....
strcpy(sir, "alfa");
```

**secvența 2**

#### Temă



Scrieți trei variante de programe (folosind cele trei metode de prelucrare a șirurilor de caractere) prin care copiați într-un șir de caractere primele **n** caractere dintr-un al doilea șir de caractere. În cazul în care nu folosiți funcția de sistem, implementarea se va face cu ajutorul subprogramelelor. Cele două șiruri de caractere și numărul de caractere **n** se transmit ca parametri. În programul principal se citește de la tastatură șirul de caractere din care se copiază și numărul de caractere care se copiază și se afișează șirul de caractere obținut.

## Concatenarea a două șiruri de caractere

Se adaugă la sfârșitul unui șir de caractere (**șirul destinație**) conținutul unui alt șir de caractere (**șirul sursă**). Puteți să folosiți următoarele funcții de sistem (parametrii funcțiilor sunt: **s2** – șirul sursă, **s1** – șirul destinație și **n** numărul de caractere care se adaugă). Ambele funcții furnizează ca rezultat un pointer care indică adresa șirului destinație.

| Funcția          | Sintaxa apelului        | Realizează                                                                                                                                                                                                                                                                                                                    |
|------------------|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strcat()</b>  | <b>strcat(s1,s2)</b>    | Sunt adăugate din șirul sursă <b>s2</b> în șirul destinație <b>s1</b> toate caracterele, inclusiv caracterul NULL                                                                                                                                                                                                             |
| <b>strncat()</b> | <b>strncat(s1,s2,n)</b> | Sunt adăugate din șirul sursă <b>s2</b> în șirul destinație <b>s1</b> maxim <b>n</b> caractere, începând cu primul caracter. Funcția adaugă la sfârșitul caracterelor adăugate caracterul NULL. În cazul în care <b>n</b> este mai mare decât lungimea șirului sursă, se va adăuga tot șirul sursă, dar nu și alte caractere. |

### Exemplu

```
#include <iostream.h>
#include <string.h>
void main()
{char sir1[256], sir2[256];
 cout<<"Primul sir de caractere "; cin.get(sir1,100); cin.get();
 cout<<"Al doilea sir de caractere "; cin.get(sir2,100);
 strcat(sir1,sir2); cout<<sir1;}
```

### Temă



Scrieți trei variante de program (folosind cele trei metode de prelucrare a șirurilor de caractere) prin care concatenați la un șir de caractere primele **n** caractere din cel de al doilea șir de caractere. În cazul în care nu folosiți funcția de sistem, implementarea se va face cu ajutorul subprogramelor. Cele două șiruri de caractere și numărul de caractere **n** se transmit ca parametri. În programul principal se citesc de la tastatură șirurile de caractere care se concatenează și numărul de caractere **n** și se afișează șirul de caractere obținut.

### Atenție

În prelucrările prin care adăugați sau copiați un șir de caractere sursă la un șir de caractere destinație trebuie să aveți grijă **să nu depășiți locațiile de memorie rezervate șirului destinație**, deoarece este posibil ca sistemul de operare să nu sesizeze această eroare și să scrie în zona rezervată altor variabile de memorie.

### Temă



Urmăriți instrucțiunile următorului program. Ce ar trebui să afișeze? Executați programul. Ce constatați? Explicați de ce au fost afișate aceste valori pentru rezultate.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir1[4]="alfa",sir2[4]; cout<<sir1<<" "<<&sir1<<" "<<&sir2<<endl;
 strcpy(sir2,"alfabet"); cout<<sir1<<" "<<sir2;}
```

**Observație:** Funcțiile prin care se scriu caractere într-un șir destinație (copierea unui șir și concatenarea a două șiruri) adaugă caracterul NULL la sfârșitul șirului destinație, degrevând programatorul de această sarcină.

## Compararea a două șiruri de caractere

Compararea a două șiruri de caractere se face prin compararea codului ASCII al caracterelor din aceeași poziție a fiecărui șir. Dacă cele două șiruri nu au aceeași lungime, șirul cu lungime mai mică este completat la sfârșit, până la egalarea lungimilor, cu caracterul NULL care are codul ASCII 0. Operația de comparare începe cu prima poziție din șir și continuă cu următoarele poziții numai dacă pozițiile anterioare sunt identice în ambele șiruri. De exemplu, șirul de caractere **Idee** este mai mare decât șirul de caractere **IDei** deoarece, în poziția a doua, caracterele din cele două șiruri nu mai sunt identice, iar codul ASCII al caracterului **d** este mai mare decât codul ASCII al caracterului **D**. Operația de comparare se oprește după cel de al doilea caracter și nu mai contează codurile caracterelor din poziția patru, care sunt diferite pentru **e** și pentru **i**.

Puteți să folosiți următoarele funcții de sistem (toate funcțiile furnizează un rezultat de tip **int**; parametrii funcțiilor sunt: **s1** și **s2** – șirurile de caractere care se compară și **n** numărul de caractere care se compară):

| Funcția           | Sintaxa apelului         | Realizează                                                                                                                                                                                                                   |
|-------------------|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strcmp()</b>   | <b>strcmp(s1,s2)</b>     | Compară cele două șiruri de caractere. Dacă sunt identice, rezultatul este 0. Dacă <b>s1</b> este mai mare decât <b>s2</b> , rezultatul este pozitiv. Dacă <b>s1</b> este mai mic decât <b>s2</b> , rezultatul este negativ. |
| <b>stricmp()</b>  | <b>stricmp(s1,s2)</b>    | Compară cele două șiruri de caractere la fel ca și funcția <b>strcmp()</b> , dar fără să facă diferența între literele mari și literele mici.                                                                                |
| <b>strncmp()</b>  | <b>strncmp(s1,s2,n)</b>  | Compară primele <b>n</b> caractere din cele două șiruri de caractere, furnizând rezultatul la fel ca și funcția <b>strcmp()</b> .                                                                                            |
| <b>strncmpi()</b> | <b>strncmpi(s1,s2,n)</b> | Compară cele două șiruri de caractere la fel ca și funcția <b>strncmp()</b> , dar fără să facă diferența între literele mari și literele mici.                                                                               |

### Temă



1. Scrieți trei variante de program (folosind cele trei metode de prelucrare a șirurilor de caractere) prin care comparați două șiruri de caractere. În cazul în care nu folosiți funcția de sistem, implementarea se va face cu ajutorul subprogramelor. Cele două șiruri de caractere se transmit ca parametri. În programul principal se citesc de la tastatură șirurile de caractere care se compară și se afișează rezultatul comparației.
2. Precizați rezultatul furnizat de următoarele funcții: **strcmp("ab","a")**, **strcmp("ab","abc")**, **strcmp("ab","Ab")**, **stricmp("Abcd","Abcd")**, **stricmp("abcd","Abcd")**, **strcmp("ab","Abc")**, **strncmpi("abcd","Abcd",2)**, **strncmpi("abc","Abc",4)**, **strcmp("ab","Ab")**, **stricmp("ab","Ab")**, **strcmp("ab","A")**, **stricmp("a","A")**, **strcmp("abc","abc")**, **strncmp("abc","abcd",3)**.
3. Se citesc de la tastatură două șiruri de caractere **s1** și **s2**. Scrieți un subprogram recursiv care să compare cele două șiruri de caractere și care va furniza următoarele valori: -1 dacă **s1** este mai mic decât **s2**, 0 dacă sunt egale și +1 dacă **s1** este mai mare decât **s2**. Folosiți subprogramul recursiv pentru a compara cele două șiruri de caractere.

Compararea a două șiruri de caractere este utilă atunci când trebuie să ordonați alfabetic o mulțime de șiruri de caractere (de exemplu, o mulțime de cuvinte). Pentru rezolvarea acestui gen de probleme veți forma un vector cu elemente șiruri de caractere și veți aplica pe acest vector unul dintre algoritmii de sortare învățați.

## Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi funcțiile care prelucreză două șiruri de caractere.

**Enunțul problemei:** Să se ordoneze alfabetic o mulțime de  $n$  cuvinte citite de la tastatură.

Se consideră că un cuvânt poate avea maxim 25 de caractere, iar mulțimea – maxim 50 de cuvinte. Mulțimea de cuvinte va fi implementată printr-o matrice de caractere cu 50 de linii și 25 de coloane – `char sir[50][25]`, care poate fi considerată ca un vector ale cărui elemente sunt șiruri de caractere: primul indice precizează numărul de șiruri de caractere memorate în vector, iar al doilea indice, lungimea maximă a fiecărui șir de caractere. Așadar, matricea fiind un vector de cuvinte, fiecare linie a matricei – `sir[i]` – va memora un cuvânt (un șir de caractere cu maxim 25 de caractere) din mulțimea de cuvinte. Se va folosi pentru sortarea cuvintelor metoda selecției directe. Variabila intermediară `aux` – prin intermediul căreia se face interschimbarea între cele două elemente ale vectorului de cuvinte – este de tip șir de caractere cu maxim 25 de caractere.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[50][25],aux[25]; int n,i,j;
 cout<<"Numarul de cuvinte= "; cin>>n; cin.get();
 for (i=0;i<n;i++)
 {cout<<"cuvantul: "; cin.get(sir[i],25); cin.get();}
 for (i=0;i<n-1;i++)
 for (j=i+1;j<n;j++)
 if (strcmp(sir[i],sir[j])>0)
 {strcpy(aux,sir[i]); strcpy(sir[i],sir[j]); strcpy(sir[j],aux);}
 for (i=0;i<n;i++) cout<<sir[i]<<endl; }
```

### Temă



1. Refaceți programul anterior folosind sortarea prin metoda bulelor.
2. Se consideră  $n$  mulțimi de cuvinte  $A_i$ , fiecare mulțime având  $n_i$  cuvinte. Să se genereze toate textele de  $n$  cuvinte care se pot forma cu cuvintele din aceste mulțimi, cuvântul  $i$  din text aparținând mulțimii  $A_i$ .
3. Se citesc de la tastatură două șiruri de caractere  $s_1$  și  $s_2$ . Scrieți un subprogram recursiv care să verifice dacă șirul  $s_1$  este anagrama șirului  $s_2$ .



### Observație

În marea majoritate a limbajelor de programare, pentru a manipula șiruri de caractere, există implementat un tip special de date. De obicei, prin această implementare, șirul de caractere poate avea o lungime de maxim 255 de caractere (ocupând 256 de octeți, din care unul se folosește pentru a memora lungimea șirului de caractere). Pe acest tip de dată se pot aplica operatorul de concatenare, operatorul de atribuire și operatorii relaționali. În limbajul C++ șirul de caractere nu este implementat ca un tip de dată și nu pot fi aplicați operatorii menționați. Realizarea acestor operații se poate face folosind funcțiile de sistem, astfel:

| Operatorul     | Operația                                                                                           | Funcția                     |
|----------------|----------------------------------------------------------------------------------------------------|-----------------------------|
| de atribuire   | $s_1 \leftarrow s_2$                                                                               | <code>strcpy(s1, s2)</code> |
| de concatenare | $s_1 + s_2$                                                                                        | <code>strcat(s1, s2)</code> |
| relațional     | $s_1 = s_2$ sau $s_1 <> s_2$ sau $s_1 > s_2$ sau $s_1 < s_2$ sau $s_1 \geq s_2$ sau $s_1 \leq s_2$ | <code>strcmp(s1, s2)</code> |

### 2.3.3.2. Prelucrarea unui șir de caractere

Pentru prelucrarea unui șir de caractere puteți folosi următoarele operații:

- inițializarea unui șir de caractere cu același caracter;
- inversarea conținutului unui șir de caractere;
- transformări între literele mari și literele mici din șir;
- căutarea unui caracter într-un șir:
  - găsirea primeia sau a ultimei apariții în șir a caracterului;
  - furnizarea poziției primeia sau a ultimei apariții în șir a caracterului;
  - numărarea aparițiilor unui caracter într-un șir.

#### Inițializarea unui șir de caractere cu același caracter

Prin această operație se atribuie pozițiilor dintr-un șir aceeași valoare – pe o lungime comunicată prin program. Vectorul de caractere se parcurge cu ajutorul indicilor:

```
#include <iostream.h>
void main()
{char sir[256],c; int i,n;
 cout<<"Numarul de caractere"; cin>>n;
 cout<<"Caracterul de umplere: "; cin>>c;
 for (i=0;i<n;i++) sir[i]=c; sir[i]=0; cout<< sir;}
```

În cazul în care șirul de caractere are deja o valoare, vechea valoare poate fi modificată prin înlocuirea tuturor caracterelor cu un același caracter:

```
#include <iostream.h>
void main()
{char sir[256],c; int i; cout<<"Textul: "; cin.get(sir,100);
 cout<<"Caracterul de umplere: "; cin>>c;
 for (i=0;sir[i];i++) sir[i]=c; sir[i]=0; cout<< sir;}
```

#### Temă



Rescrieți programele anterioare parcurgând vectorul cu ajutorul pointerilor.

Puteți să folosiți următoarele funcții de sistem (parametrii funcțiilor sunt: **sir** – șirul de caractere, **ch** – caracterul și **n** numărul de caractere). Ambele funcții furnizează ca rezultat un pointer care indică adresa șirului.

| Funcția          | Sintaxa apelului         | Realizează                                                                                                                                                                                                                                                                                                               |
|------------------|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strset()</b>  | <b>strset(sir,ch)</b>    | Șirul <b>sir</b> este parcurs începând cu primul caracter, până la sfârșitul lui, fiecare caracter fiind înlocuit cu caracterul <b>ch</b> , mai puțin caracterul NULL.                                                                                                                                                   |
| <b>strnset()</b> | <b>strnset(sir,ch,n)</b> | În șirul <b>sir</b> sunt parcurse primele <b>n</b> caractere, începând cu primul caracter, dar nu mai mult decât lungimea șirului, fiecare caracter fiind înlocuit cu caracterul <b>ch</b> . Dacă <b>n</b> este mai mare sau egal cu lungimea șirului <b>sir</b> , funcția va avea același efect ca și <b>strset()</b> . |

#### Temă



Rescrieți programele anterioare folosind funcțiile de sistem corespunzătoare.

## Inversarea conținutului unui șir de caractere

Se inversează conținutul unui vector de caractere în el însuși, mai puțin caracterul NULL. Pentru rezolvarea acestei probleme se poate folosi funcția sistem **strrev()** care are sintaxa: **strrev(sir)**, unde **sir** este șirul care se inversează. Funcția furnizează ca rezultat un pointer care indică adresa șirului inversat.

### Temă



Scrieți trei variante de program (folosind cele trei metode de prelucrare a șirurilor de caractere) prin care inversați un șir de caractere citit de la tastatură, în două variante:

- într-un alt șir de caractere;
- în el însuși.

## Transformarea literelor mari în litere mici și invers

Prin această operație, caracterele scrise cu litere mari dintr-un text sunt transformate în litere mici, și invers, prin modificarea codului ASCII al caracterului. Se va ține cont că, pentru un caracter literă, diferența dintre codul ASCII al literei mari și codul ASCII al literei mici este 32. Pentru transformare se mai pot folosi funcțiile **tolower()** și **toupper()** cu sintaxa: **tolower(ch)** – transformă caracterul **ch** din literă mare în literă mică; altfel, îl lasă neschimbat, și respectiv **toupper(ch)** – transformă caracterul **ch** din literă mică în literă mare; altfel, îl lasă neschimbat. Ambele funcții furnizează un rezultat de tip **char** și au nevoie de fișierul antet **<ctype.h>**. În exemplul următor, literele mari din șir sunt transformate în litere mici, folosind codul ASCII, și apoi literele mici sunt transformate în litere mari folosind funcția **toupper()**. Vectorul de caractere se parcurge cu ajutorul pointerului:

```
#include <iostream.h>
#include <ctype.h>
void main()
{char sir[256],*p; cout<<"sirul de caractere: "; cin.get(sir,100);
//literele mari vor fi transformate în litere mici
for (p=sir;*p;p++) if (*p>='A'&& *p<='Z') *p+=32; cout<<sir<<endl;
//literele mici vor fi transformate în litere mari
for (p=sir;*p;p++) *p=toupper(*p); cout<<sir;}
```

### Temă



Rescrieți programul anterior parcurgând vectorul cu ajutorul indicilor; transformați mai întâi literele mici în litere mari folosind codul ASCII și apoi literele mari în litere mici folosind funcția **tolower()**.

Se mai pot folosi funcțiile sistem care au parametrul **sir** – șirul de caractere care se transformă. Ambele funcții furnizează ca rezultat un pointer care indică adresa șirului transformat.

| Funcția         | Sintaxa apelului   | Realizează                                                                                                        |
|-----------------|--------------------|-------------------------------------------------------------------------------------------------------------------|
| <b>strlwr()</b> | <b>strlwr(sir)</b> | În șirul de caractere <b>sir</b> transformă literele mari în litere mici. Restul caracterelor nu sunt modificate. |
| <b>strupr()</b> | <b>strupr(sir)</b> | În șirul de caractere <b>sir</b> transformă literele mici în litere mari. Restul caracterelor nu sunt modificate. |

## Căutarea unui caracter într-un șir

Prin această operație se parcurge șirul de caractere pentru a verifica dacă există caracterul căutat. Se pot folosi funcții sistem care au parametrii: **sir** – șirul de caractere în care se caută, și **ch** – caracterul care se caută. Ambele funcții furnizează ca rezultat un pointer care indică poziția caracterului căutat, dacă l-au găsit; altfel, dacă nu l-au găsit, returnează pointerul caracterului NULL care marchează sfârșitul șirului de caractere.

| Funcția          | Sintaxa apelului        | Realizează                                                                                                                                        |
|------------------|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strchr()</b>  | <b>strchr(sir, ch)</b>  | Furnizează ca rezultat un pointer către prima apariție a caracterului <b>ch</b> în șirul de caractere <b>sir</b> (cea din extremitatea stângă).   |
| <b>strrchr()</b> | <b>strrchr(sir, ch)</b> | Furnizează ca rezultat un pointer către ultima apariție a caracterului <b>ch</b> în șirul de caractere <b>sir</b> (cea din extremitatea dreaptă). |

**Observație.** Parcurgerea șirului de caractere se poate face de la începutul șirului de caractere către sfârșitul lui, găsindu-se astfel prima apariție, sau de la sfârșitul șirului de caractere către începutul său, găsindu-se ultima apariție. De exemplu, în șirul de caractere *ala бала portocala* prima apariție a caracterului *a* este în poziția 1 și ultima apariție a caracterului *a* este în poziția 18.

**Exemplul 1** – Se afișează numărul de apariții ale unui caracter *c* într-un șir de caractere *sir* și pozițiile în care apare caracterul în șir. Localizarea pozițiilor în care se găsește caracterul se face cu pointerul *p*.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256],c,*p; int nr=0;
 cout<<"Sirul de caractere "; cin.get(sir,100);
 cout<<"Caracterul cautat "; cin>>c;
 cout<<"caracterul "<<c<<" apare in pozitiile:"<<endl;
 p=strchr(sir,c); //se caută prima apariție
 while (p) //cât timp se găsește caracterul
 {cout<<p-sir+1<<" "; nr++; p=strchr(p+1,c);}
 /*se caută următoarea apariție a caracterului începând cu
 poziția imediat următoare celei în care a fost găsit */
 cout<<endl<<"apare in "<<nr<<" pozitii";}
```

**Exemplul 2** – Se determină numărul de apariții ale caracterului *c* în șirul de caractere *sir*. Subprogramul **numar()** numără aceste apariții (*c* și *sir* se transmit prin parametri).

```
#include <iostream.h>
#include <string.h>
int numar(char *s, char c)
{int n=0; s=strchr(s,c);
 while (s) {n++; s=strchr(s+1,c);}
 return n;}
void main()
{char sir[256],c; cout<<"sirul de caractere= "; cin.get(sir,255);
 cout<<"caracterul= "; cin>>c;
 cout<<"caracterul "<<c<<" apare in sirul "<<sir;
 cout<<" de "<<numar(sir,c)<<" ori";}
```

**Exemplul 3** – Se numără cuvintele dintr-un text. Se consideră că separarea cuvintelor se face printr-un singur spațiu. Pentru aceasta, se folosesc doi pointeri *p* și *q* în care se memorează adresa de la care începe cuvântul, respectiv adresa la care se termină cuvântul. Adresa la care se termină cuvântul este adresa la care se găsește primul spațiu din restul textului. Pointerul *q* este folosit pentru a ne deplasa în text pe primul spațiu care urmează după adresa memorată în pointerul *p*.



```
#include <iostream.h>
#include <string.h>
void main()
{char text[256],*p,*q; int nr=1;
 cout<<"Textul: "; cin.get(text,256); q=strchr(text, ' ');
 //în pointerul q se memorează adresa primului spațiu
 while (q) //cât timp se mai găsește un spațiu în text
 {nr++; p=q+1; //pointerul p este poziționat după spațiul găsit
 q=strchr(p, ' ');}
 //în pointerul q se memorează adresa primului spațiu găsit
 //începând de la adresa memorată în pointerul p
 cout<<"S-au găsit "<<nr<<" cuvinte";}
```

### Temă



1. Rescrieți programul de la Exemplul 2 folosind funcția **strchr()**.
2. Rescrieți programele anterioare fără să folosiți funcții de sistem, parcurgând vectorul de caractere cu ajutorul indicilor, respectiv cu ajutorul pointerilor.
3. Scrieți trei variante de programe (folosind cele trei metode de prelucrare a șirurilor de caractere) prin care să afișați prima apariție și ultima apariție a unui caracter **c** într-un șir de caractere **sir**.
4. Scrieți un program care să afișeze numărul de apariții ale unui caracter **c** într-un șir de caractere **sir** și pozițiile în care apare caracterul în șir.
5. Scrieți un program care să afișeze ordinea în care apar într-un șir două caractere, **c1** și **c2**. **Indicație.** Se folosesc doi pointeri, **p** (pentru localizarea caracterului **c1**) și **q** (pentru localizarea caracterului **c2**). Stabilirea ordinii de afișare se face prin compararea celor doi pointeri (a adreselor memorate în pointeri).
6. Scrieți un program care să afișeze numărul de caractere între primele două apariții, într-un șir de caractere ale unui caracter **c** precizat. **Indicație.** Se folosesc doi pointeri, **p** și **q**, către elementele șirului. Adresa primei apariții a caracterului **c** se memorează în pointerul **q**, iar adresa următoarei apariții în pointerul **p**. Numărul de caractere dintre cele două apariții se calculează ca diferență dintre adresele memorate în pointerii **p** și **q**.

### Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi funcțiile care prelucrează un șir de caractere.

**Enunțul problemei 1:** Se citește de la tastatură un text în care cuvintele sunt separate printr-un singur spațiu. Să se afișeze cuvintele din text care sunt palindrom.

Se folosesc următoarele șiruri de caractere: **text** pentru a citi textul de la tastatură, **cuv** pentru a extrage un cuvânt din text și **inv** pentru a inversa caracterele cuvântului. Pointerii **p** și **q** se folosesc pentru a memora adresa de la care începe cuvântul, respectiv adresa la care se termină cuvântul. Diferența dintre pointeri (**p-q**) reprezintă numărul de caractere din text care formează un cuvânt, începând de la adresa **p**.

```
#include <iostream.h>
#include <string.h>
void main()
{char text[256],cuv[25],inv[25],*p,*q;
 cout<<"Textul: "; cin.get(text,256); p=text; q=strchr(text, ' ');}
```

```

while (q)
{ *cuv=NULL; //variabila pentru cuvânt se inițializează cu șirul vid
 strncat(cuv,p,q-p); //se obține cuvântul prin concatenarea la
 // șirul vid cuv a p-q caractere din șirul care începe la adresa p
 strcpy(inv,cuv); //se copiază cuvântul cuv în șirul inv
 strrev(inv); //se inversează cuvântul (șirul de caractere inv)
 if (!strcmp(cuv,inv)) cout<<cuv<<endl;
 //se compară cele două șiruri de caractere (cuv și inv);
 //dacă sunt egale, se afișează cuvântul
 p=q+1; // pointerul p este poziționat după spațiul găsit
 //în pointerul q se memorează adresa primului spațiu găsit
 //începând de la adresa memorată în pointerul p
 q=strchr(p, ' ');
 //se extrage și se prelucrează ultimul cuvânt:
 *cuv=NULL; strncat(cuv,p,q-p); strcpy(inv,cuv); strrev(inv);
 if (!strcmp(cuv,inv)) cout<<cuv; }

```

### Temă



De ce, pentru a extrage cuvântul, s-a folosit funcția pentru concatenarea a două șiruri de caractere **strncat**(cuv,p,q-p) și nu funcția pentru copierea unui șir de caractere **strcpy**(cuv,p,q-p)?

**Enunțul problemei 2:** Se citește un cuvânt de la tastatură. Să se traducă într-o „limbă păsărească” definită astfel:

- după fiecare vocală se adaugă litera **m** urmată de vocala respectivă;
- dacă ultima literă din cuvânt este o consoană, se adaugă la sfârșitul cuvântului șirul de caractere „ala”.

Se folosesc șirurile de caractere: **s1** pentru a citi cuvântul de la tastatură, **s2** pentru cuvântul tradus și **v** pentru a memora vocalele. Pointerii **p** și **q** se folosesc pentru a parcurge cuvântul citit, respectiv cuvântul tradus. Pentru a testa caracterul curent (de la adresa **p**) se folosește funcția **strchr()** prin care se caută caracterul în șirul de vocale. Dacă este găsit, la adresa **q** se adaugă caracterul, litera „m” și din nou caracterul; altfel, se adaugă numai caracterul. Pentru a verifica dacă s-a ajuns la ultimul caracter, se compară diferența dintre adresa caracterului și adresa de început a cuvântului (**p-s1**) cu lungimea cuvântului, din care se scade ultimul caracter (**strlen(s1)-1**) – pentru ultimul caracter cele două valori trebuie să fie egale.

```

#include <iostream.h>
#include <string.h>
void main()
{char s1[25],s2[75],v[]="aeiou",*p,*q; cout<<"cuvant= ";cin>>s1;
 for (p=s1,q=s2;*p;p++,q++)
 {if (strchr(v,tolower(*p))) {*q=*p;q++;*q='m';q++;*q=*p;}
 else *q=*p;
 if (p-s1==strlen(s1)-1)
 {q++; *q=0;
 if (!strchr(v,tolower(*p))) strcat(s2,"ala");}}
 cout<<s2; }

```

### Temă



Refaceți cele două probleme folosind prelucrarea șirurilor de caractere cu ajutorul indicilor.

**Enunțul problemei 3:** Se citește un text de la tastatură. Să se afișeze frecvența de apariție a literelor în text, fără a se ține cont de diferența dintre litere mari și litere mici.

În șirul de caractere **text** se citește textul care se analizează. Vectorul **v** este un vector de contoare în care se numără apariția fiecărei litere. El are 26 de elemente, fiecare element fiind un contor pentru o literă: **v[0]** pentru litera **a**, **v[1]** pentru litera **b**, **v[2]** pentru litera **c** etc. Elementele vectorului sunt inițializate cu 0. Se parcurge textul și, pentru fiecare literă, se incrementează în vectorul **v** elementul care îi corespunde. Indicele elementului se obține făcând diferența dintre literă (**text[i]**) și 65 – codul ASCII al literei **A**. Fiecare literă din text este tratată ca literă mare prin aplicarea funcției **toupper()**. Pentru a afișa frecvența de apariție a literelor în text se parcurge vectorul **v**. Dacă elementul curent este diferit de 0, înseamnă că litera asociată lui a apărut în text, și se afișează litera mică, respectiv litera mare și numărul de apariții (valoarea elementului **v[i]**). Litera se afișează astfel: folosind operatorul de conversie (**char**) se convertește codul ASCII al literei obținut prin adunarea indicelui cu 97 (codul ASCII al literei **a**), respectiv cu 65 (codul ASCII al literei **A**).

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>
void main()
{char text[256]; int i,v[26]={0}; cout<<"Textul :"; cin.get(text,255);
for (i=0; i<strlen(text); i++) v[toupper(text[i])-65]++;
for (i=0; i<26; i++)
 if (v[i]!=0) {cout<<"Litera "<<(char)(65+i)<<" sau ";
 cout<<(char)(97+i)<<" apare de ";
 cout<<v[i]<<" ori"<<endl;};}
```



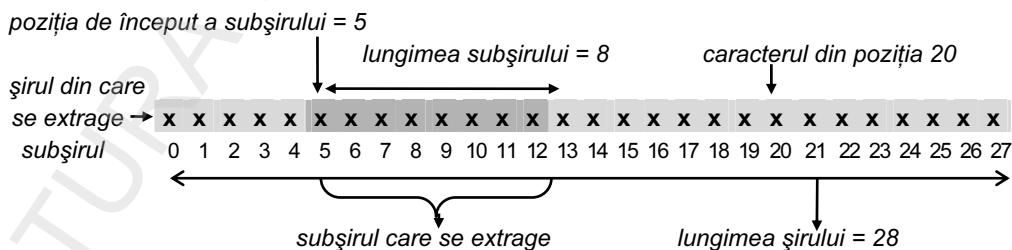
### Temă



1. Se introduce un text de la tastatură. Să se afișeze numărul literelor distincte din text și de câte ori apar ele în text. Se va ține cont de diferența dintre literele mari și literele mici.
2. Se introduce un text de la tastatură. Să se afișeze în ordine crescătoare numărul de apariții ale fiecărei litere în text. Se va preciza litera care are frecvența cea mai mare și litera care are frecvența cea mai mică. Analiza textului se va face fără să se țină cont de diferența dintre literele mari și literele mici.

### 2.3.3.3. Prelucrarea subșirurilor de caractere

Se definește **subșirul** ca fiind o porțiune dintr-un șir identificată prin poziția din care începe (**n**) și prin lungime (**m**).



De exemplu, în șirul de caractere **untdelemn**, caracterul din poziția 5 este **l**, iar subșirul **de** începe din poziția 3 și are lungimea 2.

Pentru prelucrarea subșirurilor de caractere puteți folosi următoarele operații:

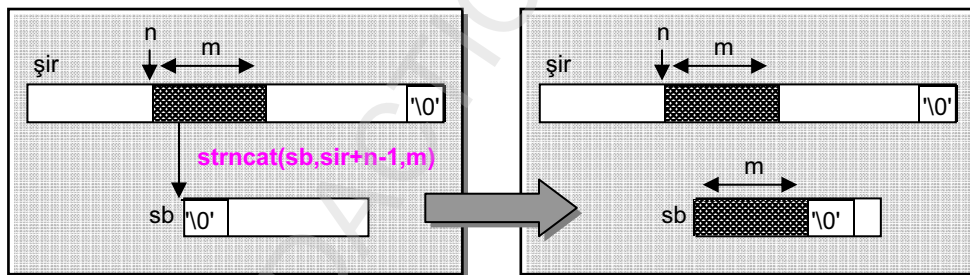
- extragerea unui subșir dintr-un șir;
- căutarea unui subșir într-un șir;
- ștergerea unui subșir dintr-un șir;
- inserarea unui subșir într-un șir;
- înlocuirea unui subșir cu un alt subșir.

### Extragerea unui subșir dintr-un șir

Prin această operație se extrage, prin copiere din șirul **sir**, un subșir care începe din poziția **n** și care are lungimea **m**. Dacă  $n > \text{strlen}(\text{sir})$ , se va extrage șirul vid. Dacă  $m > \text{strlen}(\text{sir}) - n$ , se vor extrage numai ultimele  $\text{strlen}(\text{sir}) - n$  caractere din șirul **sir**. În urma acestei operații, subșirul extras rămâne în șirul sursă.

**Exemplul 1** – Se extrage subșirul **sb** din șirul **sir** folosind funcțiile de sistem.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256],sb[50]=""; int n,m;
 cout<<"Textul "; cin.get(sir,50);
 cout<<"Pozitia din care incepe subsirul "; cin>>n;
 cout<<"Lungimea subsirului "; cin>>m;
 if (n<=strlen(sir))
 if (m>strlen(sir)-n) strcat(sb,sir+n-1);
 else strncat(sb,sir+n-1,m);
 cout<<sb; }
```



### Temă



1. Folosind programul, extrageți cuvântul *car* din cuvântul *parcare* și cuvântul *pa* din cuvântul *copac*.
2. Rescrieți programul parcurgând șirurile de caractere cu ajutorul indicilor, respectiv al pointerilor.

### Căutarea unui subșir într-un șir

Prin această operație se furnizează prima poziție din care începe în șirul **sir** un subșir **sb**. Se folosește funcția sistem **strstr()** care are sintaxa: **strstr(sir, sb)**, unde **sir** este șirul în care se caută, iar **sb** este subșirul care se caută. Dacă găsește subșirul, funcția furnizează ca rezultat un pointer către prima apariție a subșirului; altfel, furnizează valoarea NULL.

**Exemplul 1** – Se caută poziția primei apariții a subșirului **sb** în șirul **sir**. Cele două șiruri au lungimea **lg1**, respectiv **lg2**. Pentru executarea operației se parcurg șirul și subșirul de caractere folosind indicii: **i** pentru șir și **j** pentru subșir. Șirul se parcurge de la primul caracter

până la caracterul de la care, până la sfârșitul șirului, sunt mai puține caractere decât lungimea subșirului (nu mai există caractere suficiente pentru a forma împreună subșirul).

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256],sb[11]; int i,j,lg1,lg2;
 cout<<"sirul in care se cauta : "; cin.get(sir,255);cin.get();
 cout<<"subsirul care se cauta : "; cin.get(sb,10);
 lg1=strlen(sir); lg2=strlen(sb);
 for (i=0;i<lg1-lg2;i++)
 {for (j=0;sir[i+j]==sb[j] && j<lg2 ;j++);
 if (j==lg2) {cout<<"prima aparitie este in pozitia "<<i+1; i=lg1;}}
 if (i!=lg1-1) cout<<"subsirul nu există în sir";}
```

### Temă



Folosind programul, aflați din ce poziție începe cuvântul *car* în cuvântul *parcare* și cuvântul *pa* din cuvântul *copac*.

**Exemplul 2 – Se caută și se numără toate aparițiile disjuncte** ale subșirului **sb** în șirul **sir**. Se folosește funcția **strstr()**. Pointerul **p** localizează pozițiile în care se găsește subșirul **sb**. Căutarea subșirului **sb** continuă până când nu se mai găsește (pointerul **p** are valoarea NULL).

```
#include <iostream.h>
#include <string.h>
{char sir[256],sb[11],*p,*q; int nr=0;
 cout<<"sirul in care se cauta :"; cin.get(sir,255);cin.get();
 cout<<"subsirul care se cauta :"; cin.get(sb,10);
 p=strstr(sir,sb); cout<<"subsirul apare in pozitiile: ";
 while (p) {nr++; cout<<p-sir+1<<" "; p=strstr(p+1,sb);}
 cout<<endl<<"de "<<nr<<" ori"; }
```

### Temă



1. Folosind programul anterior, aflați din ce poziție începe cuvântul *car* în cuvântul *parcare* și cuvântul *pa* din cuvântul *copac*.
2. Se citesc de la tastatură două șiruri de caractere, **sir1** și **sir2**. Să se afișeze de câte ori apare șirul **sir2** în șirul **sir1** și în ce poziții. Pentru localizarea și numărarea pozițiilor veți folosi un subprogram.
3. Scrieți trei variante de programe (folosind cele trei metode de prelucrare a șirurilor de caractere) prin care să afișați prima apariție a subșirului **sb** în șirul **sir**.
4. Scrieți două versiuni pentru un program care afișează poziția ultimei apariții a unui subșir într-un șir – o versiune în care parcurgeți șirul și subșirul cu ajutorul indicilor și o versiune în care parcurgeți șirul și subșirul cu ajutorul pointerilor.
5. Scrieți două versiuni pentru un program care caută toate aparițiile disjuncte ale subșirului **sb** în șirul **sir**, fără a folosi funcția **sistem** – o versiune în care parcurgeți șirul și subșirul cu ajutorul indicilor și o versiune în care parcurgeți șirul și subșirul cu ajutorul pointerilor.

### Ștergerea unui subșir dintr-un șir

Prin această operație se șterge din șirul **sir** un subșir **sb**. După executarea operației de ștergere, noua lungime a șirului va fi **strlen(sir) - strlen(sb)**.

**Exemplul 1** – Se șterge din șirul **sir** un subșir care începe din poziția **n** și are lungimea **m**. Dacă  $n > \text{strlen}(\text{sir})$ , șirul va rămâne neschimbat. Dacă  $m > \text{strlen}(\text{sir}) - n$ , se vor șterge numai ultimele  $\text{strlen}(\text{sir}) - n$  caractere. Pentru executarea operației se parcurge șirul folosind indicii.

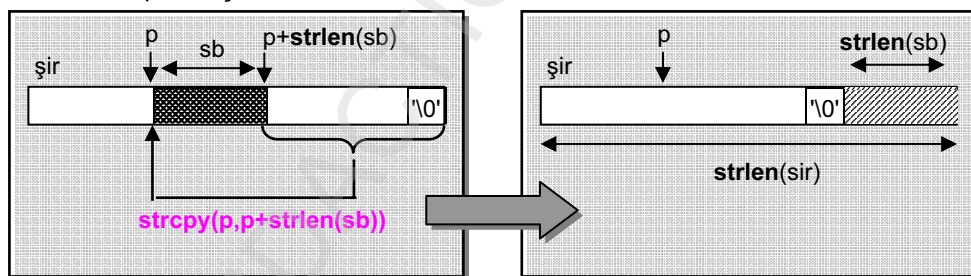
```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256]; int n,m,i;
cout<<"sirul din care se sterge :"; cin.get(sir,255);
cout<<"pozitia din care se sterge :"; cin>>n;
cout<<"numarul de pozitii care se sterg :"; cin>>m;
if (n<strlen(sir))
{if(m>strlen(sir)-n)
 for (i=n-1; sir[i+strlen(sir)-n+1]; i++)
 sir[i]=sir[i+strlen(sir)-n+1];
 else for (i=n-1; sir[i+m]; i++) sir[i]=sir[i+m];
 sir[i]=0;}
cout<<sir;}
```

### Temă



Folosind operația de ștergere a unui subșir dintr-un șir, obțineți cuvântul *pare* din cuvântul *parcare*.

**Exemplul 2** – Se șterge din șirul **sir** prima apariție a unui subșir **sb**. Pentru executarea operației se folosește funcția **strstr()** pentru a localiza prima apariție a subșirului. Dacă subșirul este găsit în șir, va fi șters prin copierea – la adresa găsită – a porțiunii din șir care urmează după subșir.



```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256],sb[11],*p;
cout<<"sirul in care se sterge :";cin.get(sir,255); cin.get();
cout<<"subsirul care se sterge :";cin.get(sb,10); p=strstr(sir,sb);
if (p) strcpy(p,p+strlen(sb)); cout<<sir;}
```

**Exemplul 3** – Se șterg din șirul **sir** toate aparițiile unui subșir **sb**. Pentru executarea operației se folosește funcția **strstr()** pentru a localiza fiecare apariție a subșirului. Când subșirul este găsit în șir, va fi șters prin copiere – la adresa la care a fost găsit – a porțiunii din șir care urmează după subșir.

```
#include <iostream.h>
#include <string.h>
```

```

void main()
{char sir[256],sb[11],*p;
 cout<<"sirul in care se sterge :";cin.get(sir,255); cin.get();
 cout<<"subsirul care se sterge :"; cin.get(sb,10); p=strstr(sir,sb);
 while (p) {strcpy(p,p+strlen(sb));p=strstr(p,sb);}
 cout<<sir;}

```

### Inserarea unui subsir într-un șir

Prin această operație se inserează subsirul **sb** în șirul **sir** în poziția **n**. După executarea operației de inserare, noua lungime a șirului va fi  $\text{strlen}(\text{sir}) + \text{strlen}(\text{sb})$ . Dacă  $\text{strlen}(\text{sir}) + \text{strlen}(\text{sb}) > \text{nmax}$  ( $\text{nmax}$  fiind lungimea fizică a șirului de caractere), se vor păstra din șirul rezultat numai  $\text{nmax}$  caractere.

#### Temă



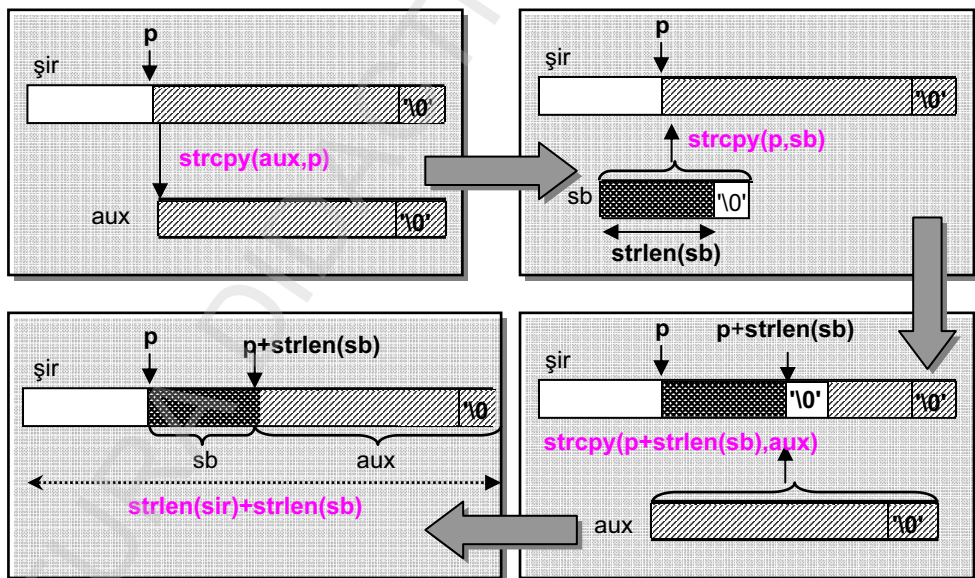
Folosind operația de inserare a unui subsir într-un șir, obțineți cuvântul *parcare* din cuvântul *pare*.

**Exemplu** – Se inserează subsirul **sb** în șirul **sir** în poziția **n**, folosind funcțiile de sistem.

```

#include <iostream.h>
#include <string.h>
void main()
{char sir[256],aux[256],sb[20],*p; int n;
 cout<<"sirul in care se insereaza:"; cin.get(sir,255);cin.get();
 cout<<"subsirul care se insereaza:"; cin.get(sb,19);
 cout<<"pozitia din care se insereaza:"; cin>>n;
 p=&sir[n-1]; strcpy(aux,p); strcpy(p,sb); strcpy(p+strlen(sb),aux);
 cout<<sir;}

```



#### Temă



Refaceți programul astfel încât să fie tratat și cazul în care  $\text{strlen}(\text{sir}) + \text{strlen}(\text{sb}) > \text{nmax}$  ( $\text{nmax}$  fiind lungimea fizică a șirului).



### Înlocuirea, într-un șir, a unui subșir cu un alt subșir

Prin această operație se înlocuiește, în șirul **sir**, subșirul **sb1** cu subșirul **sb2**. După executarea operației de înlocuire, noua lungime a șirului va fi **strlen(sir)+strlen(sb2)-strlen(sb1)**. Dacă **strlen(sir)+strlen(sb2)-strlen(sb1)>nmax** (**nmax** fiind lungimea fizică a șirului de caractere), se vor păstra din șirul rezultat numai **nmax** caractere. Pentru realizarea operației de înlocuire se execută, în ordine, următoarele operații:

- se caută în șirul **sir** subșirul **sb1**;
- se șterge subșirul **sb1** din șirul **sir**;
- se inserează în șirul **sir** subșirul **sb2** în poziția în care a fost găsit subșirul **sb1**.

#### Temă



Desenați diagrama operației de înlocuire a unui subșir într-un șir folosind modelele de diagrame de la operațiile: extragerea unui subșir dintr-un șir, ștergerea unui subșir dintr-un șir și inserarea unui subșir într-un șir.

**Exemplul 1** – Se înlocuiește, în șirul **sb**, subșirul **s1** cu subșirul **s2**, folosind funcțiile de sistem. Cele trei șiruri de caractere se citesc de la tastatură.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[256],aux[256],s1[20],s2[20],*p;
 cout<<"sirul in care se inlocuieste ":" cin.get(sir,255); cin.get();
 cout<<"subsirul care se cauta ":" cin.get(s1,19); cin.get();
 cout<<"subsirul cu care se inlocuieste:" cin.get(s2,19);
 p=strstr(sir,s1);
 while (p) {strcpy(p,p+strlen(s1)); strcpy(aux,p); strcpy(p,s2);
 strcpy(p+strlen(s2),aux); p=strstr(p,s1);}
 cout<<sir;}
```

#### Temă



Modificați programul astfel încât să fie înlocuită numai prima apariție a subșirului **s1** în șirul **sir**. Transformați cuvântul *sortare* în *serbare* folosind operația de înlocuire, într-un șir de caractere, a unui subșir cu un alt subșir, realizată de acest program.

**Exemplul 2** – Pentru rezolvarea problemei din exemplul anterior, se folosesc subprogramele: **gasit()**, pentru a verifica dacă s-a mai găsit subșirul **sb1** în șirul **sir**, **sterg()** care șterge subșirul **sb1** din șirul **sir**, și **inserez()**, care inserează subșirul **sb2**. Subprogramul **gasit()** furnizează două rezultate: unul prin numele său (este de tip logic și precizează dacă s-a găsit sau nu s-a găsit subșirul) și altul prin parametrul **p**, care este de tip pointer și care furnizează adresa subșirului (în cazul în care s-a găsit) – informația care se transmite fiind o adresă care se poate modifica în interiorul subprogramului, transferul s-a făcut prin referință.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
int gasit(char * &p, char sb[])
{p=strstr(p,sb);
 if (p==NULL) return 0; else return 1;}
void sterg(char *p, char sb[])
{strcpy(p,p+strlen(sb));}
```

```

void inserez(char *p, char sb[])
{char aux[256];
 strcpy(aux,p); strcpy(p,sb); strcpy(p+strlen(sb),aux);}

void main()
{char sir[256],sb1[20],sb2[20],*p=sir;
 cout<<"sirul de caractere= "; cin.get(sir,255); cin.get();
 cout<<"subsirul care se cauta= "; cin.get(sb1,19); cin.get();
 cout<<"subsirul care se inlocuieste= "; cin.get(sb2,19);
 while (gasit(p,sb1)) {sterg(p,sb1); inserez(p,sb2);}
 cout<<sir;}

```

### Temă



Folosind operațiile cu subșiruri, realizați următoarele transformări de cuvinte:

**car** → **caviar**

**cor** → **color**

**parcare** → **partajare**

**covor** → **cotor** → **motor** → **mosor**

**covor** → **cotor** → **color** → **cosor** → **cosar**

**cod** → **codare** → **decodare**

### Alte funcții utile în prelucrarea șirurilor de caractere

În prelucrarea șirurilor de caractere mai puteți folosi următoarele funcții (**s1** și **s2** – parametrii acestor funcții – sunt șiruri de caractere):

| Funcția          | Sintaxa apelului      | Realizează                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------------|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>strspn()</b>  | <b>strspn(s1,s2)</b>  | Furnizează ca rezultat numărul de caractere consecutive din șirul <b>s1</b> (începând cu primul caracter), care se găsesc printre caracterele din șirul <b>s2</b> .                                                                                                                                                                                                                                                                                  |
| <b>strcspn()</b> | <b>strcspn(s1,s2)</b> | Furnizează ca rezultat numărul de caractere consecutive din șirul <b>s1</b> (începând cu primul caracter), care nu se găsesc printre caracterele din șirul <b>s2</b> .                                                                                                                                                                                                                                                                               |
| <b>strpbrk()</b> | <b>strpbrk(s1,s2)</b> | Furnizează ca rezultat un pointer către primul caracter din șirul <b>s1</b> care se găsește și în șirul <b>s2</b> . Dacă nici un caracter din șirul <b>s1</b> nu se găsește printre caracterele șirului <b>s2</b> , funcția furnizează ca rezultat adresa nulă.                                                                                                                                                                                      |
| <b>strtok()</b>  | <b>strtok(s1,s2)</b>  | Șirul <b>s2</b> este un șir de caractere care pot fi folosite ca separatori, iar șirul <b>s1</b> este format din mai multe entități separate prin unul dintre separatorii din șirul <b>s2</b> . Funcția înlocuiește separatorii prin caracterul NULL și furnizează ca rezultat un pointer către primul caracter al primei entități. Pentru a găsi următoarea entitate din șirul <b>s1</b> , apelarea funcției se va face cu <b>strtok(NULL,s2)</b> . |

### Exemplu:

```

#include <iostream.h>
#include <string.h>
#include <stdio.h>
void main()
{char sir[]="Azi, Ana are mere.",sp[]=",. ",*p;
 cout<<strspn("1234567890","1D2C3B")<<endl; //afișează 3
 cout<<strspn("1234567890","2D1C3B")<<endl; //afișează 3
 cout<<strspn("1234567890","ABC")<<endl; //afișează 0
}

```

```
cout<<strcspn("1234567890","ABC457")<<endl; //afișează 3
cout<<strcspn("1234567890","ABC")<<endl; //afișează 10
cout<<strcspn("1234567890","025")<<endl; //afișează 1
p=strpbrk("1234567890","052");
if (p) cout<<"primul caracter este "<<*p<<endl; //afișează 2
p=strpbrk("1234567890","ABC");
if (p) cout<<"primul caracter este "<<*p<<endl; //nu afișează
p=strtok(sir,sp); if (p) cout<<p<<endl; //afișează Azi
p=strtok(NULL,sp); if (p) cout<<p<<endl; //afișează Ana
p=strtok(NULL,sp); if (p) cout<<p<<endl; //afișează are
p=strtok(NULL,sp); if (p) cout<<p<<endl; //afișează mere
p=strtok(NULL,sp); if (p) cout<<p<<endl; //nu afișează nimic
```

## Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi funcțiile ce prelucreează subșiruri de caractere.

**Enunțul problemei 1:** Se citește de la tastatură, ca șir de caractere, codul numeric al unei persoane. Să se afișeze următoarele informații: sexul și data de naștere ale persoanei.

Codul numeric personal este format din 13 caractere, astfel: saallzzxxxxxx, unde s precizează sexul persoanei (1 – masculin și 2 – feminin), iar aa – anul, ll – luna și zz – ziua din data de naștere. Pentru extragerea informațiilor, trebuie extrase subșirurile de caractere: s1 – sexul și s2 – anul, s3 – luna, s4 – ziua datei de naștere.

```
#include <iostream.h>
#include <string.h>
void main()
{char cn[14],s1[2]="",s2[3]="",s3[3]="",s4[3]="";
 cout<<"Codul numeric personal este "; cin.get(cn,13);
 cout<<"Persoana este de sex ";
 if (strcmp(strncat(s1,cn,1),"1")) cout<<"feminin"<<endl;
 else cout<<"masculin"<<endl;
 strncat(s2,cn+1,2); strncat(s3,cn+3,2); strncat(s4,cn+5,2);
 cout<<"si s-a nascut la data "<<s4<<" - "<<s3<<" - "<<s2;}
```

**Enunțul problemei 2:** Se citește de la tastatură un număr real sub forma unui șir de caractere. Partea întreagă este separată de partea fracționară prin virgulă. Să se afișeze partea întreagă și partea zecimală.

Pentru separarea celor două entități se folosește funcția `strtok()`:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{char numar[20],*p; cout<<"Numarul: "; cin.get(numar,19);
 p=strtok(numar,","); if (p) cout<<"partea intreaga este "<<p<<endl;
 p=strtok(NULL,","); if (p) cout<<"partea fractionara este "<<p<<endl;}
```

**Enunțul problemei 3:** Se citește un text de la tastatură. Cuvintele sunt separate prin spațiu sau caracterele: „!?”. Să se numere cuvintele din text și să se afișeze fiecare cuvânt pe un rând.

Cuvintele sunt entitățile din text care vor fi identificate folosind funcția `strtok()`:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{char text[1000],sep[]=" .,!?",*p; int n=0;
 cout<<"Textul: "; cin.get(text,999);
 p=strtok(text,sep);
 while (p) {n++; cout<<p<<endl; p=strtok(NULL,sep);}
 cout<<"S-au gasit "<<n<<" cuvinte";}
```

**Enunțul problemei 4:** Se citește un text de la tastatură. Să se afișeze câte cifre conține textul.

Textul poate fi format din mai multe subșiruri care conțin numai cifre. Se caută în șirul de caractere fiecare subșir cu cifre, folosind funcția `strpbrk()`, iar cu funcția `strspn()` se stabilește lungimea subșirului. Lungimea subșirului se va aduna la variabila `lc` în care se numără cifrele.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[100], cifre[]="1234567890",*p; int lc=0;
 cout<<"textul "; cin.get(sir,99); p=strpbrk(sir,cifre);
 while (p) {lc+=strspn(p,cifre); strcpy(p,p+strspn(p,cifre));
 p=strpbrk(p,cifre);}
 cout<<"Textul contine "<<lc<<" cifre";}
```

**Enunțul problemei 5:** Se citește de la tastatură un text care se va analiza astfel:

- dacă el conține numai cifre, se va afișa că este un număr;
- dacă el conține numai litere, se va afișa că este un cuvânt;
- dacă el conține numai cifre și litere, se va afișa: numai caractere alfanumerice;
- dacă el conține numai semne speciale, se va afișa: numai semne speciale;
- dacă el conține litere, cifre și semne speciale, se va afișa: toate tipurile de caractere.

În șirul de caractere `sir` se citește textul. Șirul de caractere `cifre` conține numai caractere cifre, iar șirul de caractere `litere` numai caractere litere mici. Deoarece prin operațiile de prelucrare șirul de caractere `sir` se va modifica, lungimea sa se va memora în variabila `lg`. În variabilele `lc` și `ll` se va memora numărul de cifre din text, respectiv numărul de litere. Folosind funcția `strspn()` se determină dacă textul este un număr sau un cuvânt, astfel: dacă lungimea subșirului din șir care conține numai cifre este egală cu lungimea șirului, atunci este număr, iar dacă lungimea subșirului din șir care conține numai litere este egală cu lungimea șirului, atunci este cuvânt. Folosind funcția `strcspn()` se determină dacă textul conține numai semne speciale, astfel: dacă lungimea subșirului de cifre care nu există în șir este egală cu numărul de cifre (10) și dacă lungimea subșirului de litere care nu există în șir este egală cu numărul de litere ale alfabetului (26), atunci textul nu conține cifre și litere. Pentru a determina dacă textul conține numai caractere alfanumerice, se numără literele și cifrele din text. Dacă numărul lor este egal cu lungimea șirului, atunci textul conține numai caractere alfanumerice.

```
#include <iostream.h>
#include <string.h>
void main()
{char sir[100],cifre[]="1234567890",litere[27],*p;
```

```

int i,lc=0,ll=0,lg;
cout<<"textul "; cin.get(sir,99); lg=strlen(sir);
for (i=0;i<26;i++) litere[i]='a'+i;
litere[i]=0; //s-au generat literele mici ale alfabetului
if (strspn(sir,cifre)==lg) cout<<"numar";
else if (strspn(strlwr(sir),litere)==lg) cout<<"cuvant";
else if (strcspn(cifre,sir)==10 && strcspn(litere,strlwr(sir))==26)
 cout<<"numai semne speciale";
else
{p=strpbrk(sir,cifre);
while (p) {lc+=strspn(p,cifre);
 strcpy(p,p+strspn(p,cifre));
 p=strpbrk(p,cifre);}
p=strpbrk(strlwr(sir),litere);
while (p) {ll+=strspn(strlwr(p),litere);
 strcpy(p,p+strspn(p,litere));
 p=strpbrk(strlwr(p),litere);}
if (lc+ll==lg) cout<<"numai caractere alfanumerice";
else cout<<"toate tipurile de caractere"; }}

```

**Enunțul problemei 6:** Se citește de la tastatură două șiruri de caractere. Să se numere câte dintre caracterele primului șir există și în al doilea șir.

Pentru a găsi fiecare caracter din primul șir (**sir1**) care există și în al doilea șir (**sir2**) se va folosi funcția **strpbrk()**.

```

#include <iostream.h>
#include <stdio.h>
void main()
{char s1[100],s2[100],*p; int nr=0;
cout<<"primul text "; cin.get(s1,100); cin.get();
cout<<"al doilea text "; cin.get(s2,100); p=strpbrk(s1,s2);
while (p) {nr++; p++; p=strpbrk(p,s2);}
cout<<"S-au gasit in primul sir "<<nr;
cout<<" caractere care exista si in al doilea sir";}

```



### Temă

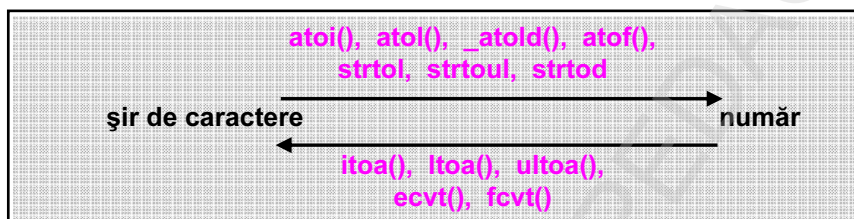


1. Executați programul de două ori: o dată pentru **sir1="alfabet"** și **sir2="alb"** și a doua oară pentru **sir2="alfabet"** și **sir1="alb"**. Ce constatați?
2. Se citește un text de la tastatură. Cuvintele se consideră separate prin spațiu, virgulă sau punct. Numărați câte cuvinte conține textul.
3. Se citește un text de la tastatură, format dintr-o singură propoziție. Se consideră că separarea cuvintelor se face prin cel puțin un spațiu. Afișați numărul de cuvinte din text, și apoi, fiecare cuvânt, pe câte un rând. Eliminați din text spațiile suplimentare și afișați textul.
4. Se citește de la tastatură un caracter **c** și apoi se introduce un text, în care separarea cuvintelor se face prin cel puțin un spațiu. Să se numere cuvintele care conțin caracterul **c**. Să se afișeze cuvintele în care apare acest caracter.
5. Se citește de la tastatură două șiruri de caractere. Să se elimine din fiecare șir caracterele care sunt comune celor două șiruri.

6. Se citesc de la tastatură trei șiruri de caractere. Să se elimine din al treilea șir caracterele care sunt comune în primele două șiruri.

### 2.3.3.4. Conversii între tipul șir de caractere și tipuri numerice

În multe probleme, datele de tip numeric trebuie transformate în date de tip șir de caractere pentru a se putea concatena, și invers, datele de tip șir de caractere care conțin numai cifre trebuie transformate în date de tip numeric pentru a se putea aplica asupra lor operatorii matematici. Pentru aceste operații de conversie se pot folosi funcțiile (toate aceste funcții sunt definite în fișierele antet `<stdlib.h>`) :



### Conversia unui șir de caractere într-un număr

Pentru acest tip de conversie se pot folosi funcții de sistem. Șirul de caractere care se convertește ar trebui să conțină numai caractere folosite pentru reprezentarea unui număr: cifre, semnul + sau -, iar pentru numerele reale, punctul ca separator între partea întreagă și partea zecimală. De exemplu, șirurile de caractere "1234" și "-12.34" conțin numai caractere folosite pentru reprezentarea unui număr, spre deosebire de șirul de caractere "12x4" ce conține litera x care nu este folosită pentru reprezentarea unui număr. Aceste funcții furnizează rezultatul astfel:

- Dacă operația de conversie se poate executa (șirul de caractere conține numai caractere folosite pentru reprezentarea unui număr), funcția va furniza o valoare numerică formată din caracterele șirului.
- Dacă șirul de caractere conține și caractere care nu sunt folosite pentru reprezentarea unui număr (de exemplu, șirul "12x34"), se va face conversia până la întâlnirea celui caracter (în exemplu, valoarea furnizată va fi 12).
- Dacă primul caracter din șir nu este un caracter permis (de exemplu, șirul "x123"), funcția va furniza valoarea 0.

Parametrii acestor funcții pot fi:

- **sir** – este de tip șir de caractere și furnizează funcției șirul de caractere care se convertește;
- **p** – este de tip `*char` (o adresă către un caracter) și va furnizează poziția primului caracter din șir care nu poate fi convertit;
- **b** – este de tip `int` și furnizează funcției baza de numerație în care trebuie să realizeze conversia (de obicei, are valoarea 10 – baza de numerație 10).

| Funcție       | Tip rezultat | Sintaxă apel     | Realizează                                                                              |
|---------------|--------------|------------------|-----------------------------------------------------------------------------------------|
| <b>atoi()</b> | int          | <b>atoi(sir)</b> | Convertește șirul de caractere <b>sir</b> într-o valoare numerică întreagă.             |
| <b>atol()</b> | long         | <b>atol(sir)</b> | Convertește șirul de caractere <b>sir</b> într-o valoare numerică întreagă de tip long. |

| Funcție          | Tip rezultat   | Sintaxă apel                 | Realizează                                                                                                                                                                                                                                                                  |
|------------------|----------------|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>atof()</b>    | double         | <b>atof(sir)</b>             | Convertește șirul de caractere <b>sir</b> într-o valoare numerică reală în virgulă mobilă dublă precizie.                                                                                                                                                                   |
| <b>_atold()</b>  | long<br>double | <b>_atold(sir)</b>           | Convertește șirul de caractere <b>sir</b> într-o valoare numerică reală în virgulă mobilă dublă precizie, de tip long.                                                                                                                                                      |
| <b>strtol()</b>  | long           | <b>strtol(sir,&amp;p,b)</b>  | Convertește șirul de caractere <b>sir</b> într-o valoare numerică întreagă de tip long. Funcției i se furnizează baza de numerație prin parametrul <b>b</b> . Funcția furnizează poziția primului caracter care nu poate fi convertit prin parametrul <b>p</b> .            |
| <b>strtoul()</b> | unsigned long  | <b>strtoul(sir,&amp;p,b)</b> | Convertește șirul de caractere <b>sir</b> într-o valoare numerică întreagă fără semn, de tip long. Funcției i se furnizează baza de numerație prin parametrul <b>b</b> . Funcția furnizează poziția primului caracter care nu poate fi convertit prin parametrul <b>p</b> . |
| <b>strtod()</b>  | double         | <b>strtod(sir,&amp;p)</b>    | Convertește șirul de caractere <b>sir</b> într-o valoare numerică reală în virgulă mobilă dublă precizie. Funcția furnizează poziția primului caracter care nu poate fi convertit prin parametrul <b>p</b> .                                                                |

### Exemplu:

```
#include <iostream.h>
#include <stdlib.h>
void main()
{char *sir="123456789",*p;
 int n1; long n2; double n3; long double n4;
 n1=atoi(sir); n2=atol(sir);
 cout<<n1<<" "<<n2<<endl; //afișează -13035 123456789
 strcpy(sir,"12345"); n1=atoi(sir); n2=atol(sir);
 cout<<n1<<" "<<n2<<endl; //afișează 12345 12345
 strcpy(sir,"12345.6789");
 n1=atoi(sir); n2=atol(sir); n3=atof(sir); n4=_atold(sir);
 cout<<n1<<" "<<n2<<" "<<n3<<" "<<n4<<endl;
 //afișează 12345 12345 12345.6789 12345.6789
 strcpy(sir,"12Aa"); n2=strtol(sir,&p,16);
 if (!*p) cout<<"conversie corecta -> "<<n2<<endl;
 else {cout<<"nu s-au putut converti decat primele ";
 cout<<p-sir<<" caractere -> "<<n2<<endl; }
 //conversie corectă -> 4778
 n2=strtol(sir,&p,10);
 if (!*p) cout<<"conversie corecta -> "<<n2<<endl;
 else {cout<<"nu s-au putut converti decat primele ";
 cout<<p-sir<<" caractere -> "<<n2<<endl; }}
 //nu s-au putut converti decat primele 2 caractere ->12
```

### Temă



În exemplele prezentate, analizați valorile afișate și explicați modul în care s-a făcut conversia în fiecare caz.



## Conversia unui număr în șir de caractere

Și pentru acest tip de conversie se pot folosi funcții de sistem. Ele furnizează ca rezultat un șir de caractere. Dacă funcția convertește un număr întreg cu semn (funcțiile `itoa()` și `ltoa()`) și numărul este negativ, semnul minus va fi scris în șirul de caractere pe prima poziție. Dacă funcția convertește un număr real (funcțiile `ecvt()` și `fcvt()`), semnul minus și punctul zecimal nu sunt scrise în șirul de caractere; informațiile despre poziția punctului zecimal și semnul numărului sunt furnizate prin intermediul unor parametri. Toate aceste funcții furnizează ca rezultat un pointer către șirul de caractere în care este convertit numărul. Parametrii acestor funcții pot fi:

- **sir** – este de tip șir de caractere și vă furnizează șirul de caractere în care este convertit numărul;
- **n** – este de tip numeric (tipul depinde de funcția folosită pentru conversie) și este furnizat funcției pentru a fi convertit;
- **b** – este de tip `int` și furnizează funcției baza de numerație în care trebuie să realizeze conversia (de obicei, are valoarea 10 – baza de numerație 10).
- **m** – este de tip `int` și furnizează funcției numărul de cifre folosite pentru conversie;
- **p** – este de tip `int` și furnizează poziția punctului zecimal față de prima poziție din șir, în cazul valorilor reale;
- **s** – este de tip `int` și vă furnizează informații despre semnul numărului în cazul valorilor reale: dacă numărul este negativ, **s** va avea valoarea 1; altfel, va avea valoarea 0.

| Funcție        | Tip număr                  | Sintaxă apel                          | Realizează                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------|----------------------------|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>itoa()</b>  | <code>int</code>           | <code>itoa(n,sir,b)</code>            | Convertește în șirul de caractere <b>sir</b> o valoare numerică întreagă <b>n</b> exprimată în baza de numerație <b>b</b> .                                                                                                                                                                                                                                                                                                                                                     |
| <b>ltoa()</b>  | <code>long</code>          | <code>ltoa(n,sir,b)</code>            | Convertește în șirul de caractere <b>sir</b> o valoare numerică întreagă de tip <code>long</code> <b>n</b> exprimată în baza de numerație <b>b</b> .                                                                                                                                                                                                                                                                                                                            |
| <b>ultoa()</b> | <code>unsigned long</code> | <code>ultoa(n,sir,b)</code>           | Convertește în șirul de caractere <b>sir</b> o valoare numerică întreagă fără semn, de tip <code>long</code> <b>n</b> , exprimată în baza de numerație <b>b</b> .                                                                                                                                                                                                                                                                                                               |
| <b>ecvt()</b>  | <code>double</code>        | <code>ecvt(n,m,&amp;p, &amp;s)</code> | Convertește într-un șir de caractere o valoare numerică reală în virgulă mobilă dublă precizie <b>n</b> . Parametrul <b>m</b> precizează numărul de caractere ale șirului; dacă numărul are mai puține cifre decât <b>m</b> , se va completa la dreapta cu caracterul cifra 0, până se obțin cele <b>m</b> caractere. Funcția furnizează prin numele ei adresa șirului de caractere, prin parametrul <b>p</b> – poziția punctului zecimal și prin parametrul <b>s</b> – semnul. |
| <b>fcvt()</b>  | <code>double</code>        | <code>fcvt(n,m,&amp;p, &amp;s)</code> | Convertește într-un șir de caractere o valoare numerică reală în virgulă mobilă dublă precizie <b>n</b> , la fel ca funcția <b>ecvt()</b> – cu deosebirea că parametrul <b>m</b> precizează numărul de caractere ale părții zecimale din șir; dacă numărul are mai puține cifre în partea fracționară decât <b>m</b> , se va completa la dreapta cu caracterul cifra 0 – până se obțin cele <b>m</b> caractere pentru partea fracționară.                                       |

Exemplu:

```
#include <iostream.h>
#include <stdlib.h>
```

```

void main()
{char sir[25],*sc; double n; int n1=-12345,m,p,s;
 long n2=213456789; unsigned long n3=4223456789;
 itoa(n1,sir,10); cout<<n1<<" "<<sir<<endl; //afișează -12345 -12345
 n1=12345; itoa(n1,sir,10); cout<<n1<<" "<<sir<<endl; //12345 12345
 ltoa(n2,sir,10); cout<<n2<<" "<<sir<<endl; //213456789 213456789
 ultoa(n3,sir,10); cout<<n3<<" "<<sir<<endl; //4223456789 4223456789
 n=9.87654; m=10; sc=ecvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //9876540000 1 0
 n=-123.456; m=15; sc=ecvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //1234560000000000 3 1
 n=9988.76; m=5; sc=ecvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //99888 4 0
 n=987.654; sc=fcvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //98765400 3 0
 n=9.87654; m=10; sc=fcvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //98765400000 1 0
 n=-123.456; m=5; sc=fcvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //12345600 3 1
 n=-1.23456; m=10; sc=fcvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; //12345600000 1 1
 n=9.87654; m=3; sc=fcvt(n,m,&p,&s);
 cout<<sc<<" "<<p<<" "<<s<<endl; } //9877 1 0

```

### Temă



1. Folosind operația de conversie, concatenați vârsta furnizată de data **v**, de tip **unsigned int**, cu șirul de caractere "ani".
2. Din exemplele prezentate, stabiliți regula folosită de funcțiile **ecvt()** și **fcvt()** pentru a rotunji numărul convertit în șir de caractere, atunci când numărul pozițiilor din șir este mai mic decât numărul de cifre din valoarea numerică.

## Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi funcțiile care fac conversia între valori numerice și șiruri de caractere.

**Enunțul problemei 1:** Se citește de la tastatură un text care poate conține și numere. Să se afișeze suma acestor numere, chiar dacă fac parte din cuvinte.

Cu funcția **strpbrk()** se identifică fiecare cifră, cu funcția **strspn()** se extrage din șirul *numar* subșirul de cifre care începe cu acea cifră (subșir care reprezintă un număr), cu funcția **strtoul()** se convertește șirul de caractere *numar* într-o valoare întreagă care se adună la suma *s*, după care se elimină din șirul inițial *sir* numărul, cu funcția **strcpy()**.

```

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{char sir[100],numar[10],cifre[]="1234567890",*p,*q,*r;
 unsigned long s=0; cout<<"textul "; cin.get(sir,100);
 p=strpbrk(sir,cifre);
 while (p)

```

```
{strcpy(numar,""); q=p+strspn(p,cifre); strncat(numar,p,q-p);
s+=strtoul(numar,&r,10); strcpy(p,q); p=strpbrk(p,cifre);}
cout<<"suma= "<<s; }
```

**Enunțul problemei 2:** Să se afișeze suma a două numere hexazecimale citite de la tastatură.

Cele două numere se citesc în șirurile de caractere **nr1** și **nr2**. Pentru fiecare număr se verifică dacă el poate reprezenta un număr în baza 16, astfel: se convertește șirul de caractere în valoarea numerică **n1**, respectiv **n2**, și se verifică dacă s-a executat corect conversia – comparând lungimea șirului **nr1**, respectiv **nr2**, cu numărul de caractere care au fost convertite (**p-nr1** sau **p-nr2 - p** fiind un pointer către poziția caracterului care nu a putut fi convertit). Suma celor două valori numerice **n1** și **n2** reprezentate în hexazecimal se convertește, cu funcția **ultoa()**, în șirul de caractere **suma** care se afișează.

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{char nr1[10],nr2[10],suma[10],*p; unsigned long n1,n2;
cout<<"primul numar "; cin.get(nr1,10); cin.get();
n1=strtoul(nr1,&p,16);
while (p-nr1!=strlen(nr1))
{cout<<"primul numar "; cin.get(nr1,10); cin.get();
n1=strtoul(nr1,&p,16);}
cout<<"al doilea numar "; cin.get(nr2,10); cin.get();
n2=strtoul(nr2,&p,16);
while (p-nr2!=strlen(nr2))
{cout<<"al doilea numar "; cin.get(nr2,10); cin.get();
n2=strtoul(nr2,&p,16);}
ultoa(n1+n2,suma,16); cout<<"suma= "<<suma; }
```

**Enunțul problemei 3:** Se citește un număr real. Să se convertească într-un șir de caractere ce va fi prelucrat astfel încât numărul real să poată fi afișat într-un format în care partea întreagă este separată de partea fracționară prin virgulă.

Numărul se citește în variabila **nr**. În șirul **nr1** se convertește cu funcția **ecvt()** valoarea numerică **nr**, iar în șirul **nr2** se obține valoarea numerică prelucrată prin adăugarea semnului minus (dacă numărul este negativ), inserarea virgulei între partea întreagă și partea fracționară și eliminarea zerourilor nesemnificative din partea zecimală a numărului.

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
void main()
{char nr1[20],nr2[20]="",*q; double nr; int p,s;
cout<<"numarul "; cin>>nr; strcpy(nr1,ecvt(nr,20,&p,&s));
if (s) strcat(nr2,"-"); // se adaugă semnul - dacă este cazul
strncat(nr2,nr1,p); //se concatenează partea întreagă
strcat(nr2,","); //se concatenează virgula
q=strchr(nr1+p,'0'); //se caută poziția primului 0 din partea zecimală
strncat(nr2,nr1+p,q-nr1-p); //se concatenează din partea
cout<<nr2; } //zecimală numai cifrele semnificative
```



## Temă



Se citesc două șiruri de caractere care conțin două numere reprezentate în baza 20. Afișați diferența celor două numere, în: a) baza 20; b) baza 10.

## Evaluare

## Răspundeți:

1. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char sir[20]="0123456789",aux[20],sb[3]="ab",*p; int n=4;
p=&sir[n-1]; strcpy(aux,p); strcpy(p+strlen(sb),aux);
strncpy(p,sb,strlen(sb)); cout<<sir;
```

Ce operație se execută prin această secvență de program?

2. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char sir[20]="calculator"; cout<<strchr(sir,'l')-sir;
```

3. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char *sir="alfabet",p=sir; p+=2; cout<<p;
```

4. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char s1[20]="calculator",s2[10]="lat",*p; p=strstr(s1,s2); cout<<p;
```

5. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char s1[10]="alfa",s2[5]="ALFA";
if(s1==strlwr(s2)) cout<<"siruri egale"; else cout<<"siruri inegale";
```

6. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char sir[20]="12alfa34beta56",*p;
for(p=sir;*p>='0' && *p<='9' && *p; p++); cout<<p;
```

7. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char sir[2][10]={"alfa","beta"};
if(sir[0]<sir[1]) cout<<sir[0]; else cout<<sir[1];
```

8. Se declară șirurile `char s1[5],s2[5];`. Ce se va afișa în urma execuției următoarei secvențe de program, dacă șirul `s1` are valoarea "125", iar șirul `s2` valoarea "75". Dar dacă șirul `s1` are valoarea "201", iar șirul `s2` valoarea "21":

```
if(strcmp(s1,s2)) cout<<atoi(s1); else cout<<atoi(s2);
```

9. Ce se va afișa în urma execuției următoarei secvențe de program:

```
char *sir[5]={ "abcd", "defgh", "123", "ab123", NULL}, *p=&sir[0][0];
cout<<*sir[0]<<" "<<*sir[0]++<<" "<<(*sir[0])++<<endl;
cout<<*sir[1]<<" "<<*sir[1]++<<" "<<(*sir[1])++<<endl;
cout<<*sir[2]<<" "<<*sir[2]++<<" "<<(*sir[2])++<<endl;
cout<<sir[1]<<" "<<sir[2][1]<<" "<<*(p+1)<<" "<<*(p+15);
```

10. Analizați interfața procesorului de texte Word. Identificați operațiile specifice pentru prelucrarea textelor care au fost rezolvate în acest capitol.

## Adevărat sau Fals:

1. Dacă șirul `s1` este șirul vid, instrucțiunea `strcat(s1,s2);` este echivalentă cu instrucțiunea `strcpy(s1,s2);`.

2. Dacă șirul *s1* este șirul vid, instrucțiunea `strncat(s1,s2,n)`; este echivalentă cu instrucțiunea `strncpy(s1,s2,n)`;
3. Următoarea secvență de program elimină ultimul spațiu din șirul *s*:

```
char s[100],*p=s;
while (*p!=' ' && *p) p++;
while (p) strcpy(s,p);
```

4. Următoarea secvență de program afișează "diferite" numai dacă șirul *s1* este diferit de șirul *s2* și șirul *s2* este diferit de șirul *s3*:

```
char s1[10],s2[10],s3[10];
if (strcmp(s1,s2) && strcmp(s2,s3)) cout<<"diferite";
```

5. Următoarea secvență de program afișează portocala:

```
char sir[]="ala bala portocala",*p=sir;
strcpy(p,strchr(sir,' '));
while (*p){strcpy(sir,p+1); strcpy(p,strchr(sir,' '));}; cout<<sir;
```

### Alegeți:

1. Care este declarația corectă pentru un șir de caractere:
  - a. `char *sir;`
  - b. `char sir[20];`
  - c. `char *sir[20];`
  - d. `char sir[];`
2. Care este declarația corectă pentru un șir de caractere care conține cuvântul "alfabet":
  - a. `char sir[7]="alfabet;`
  - b. `char sir[10]="alfabet;`
  - c. `char sir[]="alfabet;`
  - d. `char sir[8]="alfabet;`
3. Care este declarația corectă pentru un vector care conține 3 șiruri de caractere:
  - a. `char sir[3][2]={"ab","12"};`
  - b. `char sir[3][3]={"ab","12"};`
  - c. `char sir[3][2]={"ab","12",NULL};`
  - d. `char sir[][3]={"ab","12",NULL};`
4. Funcția prin care inversați caracterele dintr-un șir este:
  - a. `strtok()`;
  - b. `strpbrk()`;
  - c. `strfrx()`;
  - d. `strrev()`
5. Care va fi lungimea șirului *s* după execuția instrucțiunii:
 

```
for(i=0;(char)s[i]<='Z';i++) s[i]=i+65;
```

  - a. 26
  - b. 25
  - c. 27;
  - d. necunoscută
6. Care dintre următoarele funcții nu face parte din același fișier antet:
  - a. `strtod()`;
  - b. `ecvt()`
  - c. `strspn()`;
  - d. `ltoa()`
7. Care dintre următoarele secvențe de instrucțiuni afișează toate aparițiile disjuncte ale subșirului *sb* în șirul *sir*, variabila *p* fiind de tip pointer către tipul `char` și fiind inițializată cu valoarea *sir*:

a. `while (p)`  
`{p=strstr(sir,sb);`  
`if (p) cout<<p-sir+1;`  
`strcpy(sir,p);}`

c. `while (p)`  
`{p=strstr(p+1,sb);`  
`if (p) cout<<p-sir+1;}`

b. `while (p)`  
`{if (p) cout<<p-sir;`  
`strcpy(sir,p);`  
`p=strstr(sir,sb);}`

d. `while (p)`  
`{cout<<p-sir+1;`  
`p=strstr(p+1,sb);}`

## Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra **în echipă**. Fiecare miniproiect va conține:

- a. trei variante de programe (folosind cele trei metode de prelucrare a șirurilor de caractere); în cazul în care nu folosiți funcția de sistem, implementarea se va face cu ajutorul **subprogramelor**;
  - b. alte două exemple de probleme în care se vor folosi subprograme create pentru rezolvarea problemei inițiale;
  - c. un subprogram care să simuleze una dintre operațiile realizate de un procesor de texte – care se vor identifica în aplicația Word. (Exemple: deschiderea fișierului text, salvarea modificărilor în fișier, căutarea și înlocuirea unui șir de caractere, numărarea unor entități din text – caractere, cuvinte, propoziții, linii de text etc. –, transformarea literei de la începutul cuvântului în literă mare etc.)
1. Se introduc de la tastatură un cuvânt și un text. Se consideră că separarea cuvintelor în text se face prin cel puțin un spațiu. Să se șteargă un cuvânt precizat din text, astfel: **a)** numai prima apariție a cuvântului; **b)** toate aparițiile cuvântului.
  2. Se citesc două cuvinte de la tastatură. Să se verifice dacă ele au același prefix și/sau același sufix. În caz afirmativ, să se afișeze prefixul și/sau sufixul. De exemplu, cuvintele *vara* și *seara* au sufixul *ara*, cuvintele *incet* și *inca* au prefixul *inc*, iar cuvintele *derutat* și *decodat* au prefixul *de* și sufixul *at*.
  3. Se citesc două cuvinte de la tastatură. Să se verifice dacă ele sunt anagrame (conțin aceleași litere). Nu se va ține cont de diferența dintre literele mari și literele mici. De exemplu, cuvintele *arta* și *tara* sunt anagrame.
  4. Se introduc de la tastatură două cuvinte și un text. Se consideră că separarea cuvintelor în text se face prin cel puțin un spațiu. Să se înlocuiască în text primul cuvânt cu al doilea cuvânt, astfel:  
**a)** numai prima apariție a cuvântului; **b)** toate aparițiile cuvântului.
  5. Se citește de la tastatură un text care conține mai multe propoziții. Propozițiile se consideră separate prin: *.,!* sau *?*. Numărați câte propoziții conține textul și afișați:  
**a)** propoziția cu cele mai puține caractere; **b)** propoziția cu cele mai multe caractere;  
**c)** propoziția cu cele mai puține cuvinte; **d)** propoziția cu cele mai multe cuvinte.
  6. Se citește de la tastatură un număr natural *n* și apoi se introduce un text, în care separarea cuvintelor se face prin cel puțin un spațiu. Să se numere cuvintele care conțin *n* caractere. Să se afișeze numărul de ordine al acestor cuvinte în text (primul cuvânt sau al cincilea cuvânt etc.).
  7. Se introduce de la tastatură un text în care separarea cuvintelor se face prin cel puțin un spațiu. Să se afișeze cuvântul cel mai scurt și cuvântul cel mai lung precum și numerele de ordine ale acestor cuvinte.
  8. Se introduce un text de la tastatură. Să se afișeze numărul de vocale și numărul de consoane din text. Analiza se va face fără să se țină cont de diferența dintre literele mari și literele mici.
  9. Se introduce un text de la tastatură. Să se afișeze numărul de litere din text, numărul de cifre și numărul de caractere speciale. Analiza se va face fără să se țină cont de diferența dintre literele mari și literele mici.
  10. Se introduce de la tastatură un text în care separarea cuvintelor se face prin cel puțin un spațiu. Să se afișeze cuvintele care conțin o singură consoană, și în rest numai vocale (de exemplu: *apa*, *oaza*, *roua* etc.).

## 2.4. Înregistrarea

Pentru prelucrări mai complexe, este nevoie să grupați informații corelate în colecții de date numite structuri de date. Ați studiat deja o structură de date: tabloul de memorie. Acesta permite să grupați mai multe date de același tip și să le manipulați ca pe o singură variabilă de memorie.

Pentru a caracteriza un **obiect** (persoană, fenomen, proces etc.) este necesară o mulțime de proprietăți pe care o vom numi **listă de atribute**. Lista de atribute definește o întreagă **clasă de obiecte**. Fiecare obiect din această clasă de obiecte poate fi apoi descris prin atribuirea de valori atributelor.

### Studiu de caz

**Scop:** exemplificarea modului în care puteți caracteriza un obiect folosind o listă de atribute.

**Enunțul problemei:** *Să se caracterizeze situația școlară a unui elev dintr-o clasă, la o anumită disciplină, într-un semestru.*

Obiectul este situația școlară a unui elev la o anumită disciplină. Pentru a caracteriza acest obiect, sunt necesare următoarele atribute: numele și prenumele elevului, notele la acea disciplină și media. Așadar, colecția de atribute caracterizează situația oricărui elev dintr-o clasă, și va defini, la rândul ei, o clasă de obiecte. Pentru a descrie situația școlară a unui anumit elev, trebuie atribuite valori acestor atribute.

Pentru a reprezenta în calculator valorile atributelor pentru această listă, se pot folosi mai multe variabile de memorie independente, cu următoarele **tipuri de date**: șiruri de caractere de lungime 20 (pentru nume și prenume), numere întregi cu valori între 1 și 10 (pentru note) și numere reale (pentru medie).

Dacă trebuie păstrate informații despre toți elevii unei clase (spre exemplu, clasa are 25 de elevi), se va alege o structură de date de tip vector (cu 25 de elemente) în care fiecare element trebuie să conțină setul de date (nume, prenume, note și media) care se referă la un elev, deci care descrie comportamentul unui elev.

| <i>Elevul</i> | <i>nume</i> | <i>prenume</i> | <i>nota 1</i> | <i>nota 2</i> | <i>nota 3</i> | <i>media</i> |
|---------------|-------------|----------------|---------------|---------------|---------------|--------------|
| 1             | Bălașa      | Ana            | 9             | 10            | 9             | 9.33         |
| 2             | Dinu        | Mihai          | 10            | 9             | -             | 9.50         |
| ...           | ...         | ...            | ...           | ...           | ...           | ...          |

Acest set de date nu este format din date omogene și, prin urmare, nu poate fi reprezentat printr-o structură de date de tip matrice, în care o linie să conțină lista de atribute ale unui elev, iar o coloană să corespundă unui atribut. În acest caz, ar trebui create mai multe structuri de date de tip tablou de memorie: o structură de date de tip matrice cu elemente de tip **șir de caractere** cu lungimea de 20, care conține 25 de linii (câte una pentru fiecare elev) și două coloane (una pentru nume și alta pentru prenume), o structură de date de tip matrice cu elemente de tip **unsigned int**, care conține 25 de linii (câte una pentru fiecare elev) și trei coloane (câte una pentru fiecare notă, presupunând că elevul poate primi maxim 3 note – dacă a primit, de exemplu, numai două note, acestea se vor scrie în primele două coloane, iar următoarea se va completa cu 0), și un vector cu 25 de elemente de tip real, pentru medii. În timpul prelucrării, un elev se va identifica prin numărul de ordine *i*, care reprezintă numărul liniei în cele două matrice și numărul elementului în vector.



Acest model de reprezentare este foarte greoi din punct de vedere al algoritmilor de prelucrare. În acest caz, se poate folosi structura de date de tip **înregistrare**:

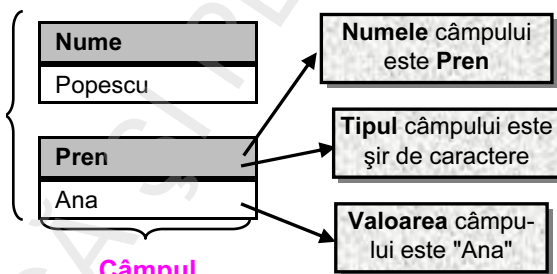


**Înregistrarea** este structura de date formată dintr-un ansamblu de date neomogene (date elementare sau structuri de date) între care există o legătură de conținut. Elementele structurii se numesc **câmpuri** și pot fi identificate după un nume.

Folosind o înregistrare, se pot păstra informații legate între ele din punct de vedere al conținutului, care sunt memorate în variabile de memorie de tipuri diferite. Legătura de conținut sau legătura logică se referă la faptul că, pentru a obține informații despre obiect, întreaga colecție de date trebuie prelucrată împreună. Fiecare descriere a listei de atribute specifice unui elev va putea fi reprezentată cu ajutorul unei înregistrări ce conține câmpurile: nume, prenume, note (trei câmpuri) și media, iar reprezentarea anterioară a elevilor dintr-o clasă va putea fi înlocuită cu o structură de date de tip vector cu 25 de elemente, ale cărui elemente sunt de tip înregistrare.

**Câmpul** este reprezentarea unui atribut din lista de atribute care descriu obiectul. El conține o dată (elementară sau o structură de date) care are legătură logică cu datele din celelalte câmpuri ale înregistrării, cu care formează împreună o entitate de informație. Fiecare câmp se identifică în listă printr-un **nume**. Câmpul este caracterizat de **tipul datei** și **valoarea datei**. De exemplu, în vectorul `clasa`, numele elevului este un câmp, prenumele alt câmp, notele alte trei câmpuri, iar media alt câmp. Fiecare câmp este caracterizat, ca și data, prin nume, tip și valoare. Accesul la elementele structurii se face prin numele câmpului. Numele câmpurilor pot să apară într-o expresie ca operanzi, la fel ca și numele datelor elementare, tipul operandului fiind determinat de tipul câmpului.

înregistrarea =  
colecția de câmpuri



Înregistrarea, ca entitate prelucrată de calculator, se identifică printr-un **nume**. De exemplu, înregistrarea care conține informații despre elev se poate identifica prin numele `elev`. Câmpurile care compun înregistrarea sunt și ele identificate prin nume: **nume** (numele), **pren** (prenumele), **nota1**, **nota2**, **nota3** (notele) și **media** (media).

| numele înregistrării | elev             |                  |                |                |                |       |
|----------------------|------------------|------------------|----------------|----------------|----------------|-------|
| numele câmpului      | nume             | pren             | nota1          | nota2          | nota3          | media |
| tipul câmpului       | șir de caractere | șir de caractere | întreg pozitiv | întreg pozitiv | întreg pozitiv | real  |
| valoarea             | Bălașa           | Ana              | 9              | 10             | 9              | 9.33  |

Identificarea unui câmp al unei înregistrări se va face folosind atât numele câmpului, cât și numele înregistrării. De exemplu, pentru a identifica câmpul **nume** din înregistrarea **elev** trebuie precizat atât numele înregistrării – **elev**, cât și numele câmpului – **nume**. Prin identificatorul **elev** se va identifica întreaga înregistrare (întreg ansamblul de câmpuri).

În clasificarea structurilor de date, înregistrarea poate fi privită ca:

- o structură de date **neomogenă** (elementele sale pot fi de tipuri diferite);
- o structură de date **internă** (utilizată ca variabilă de memorie independentă sau ca o componentă a unui tablou de memorie);

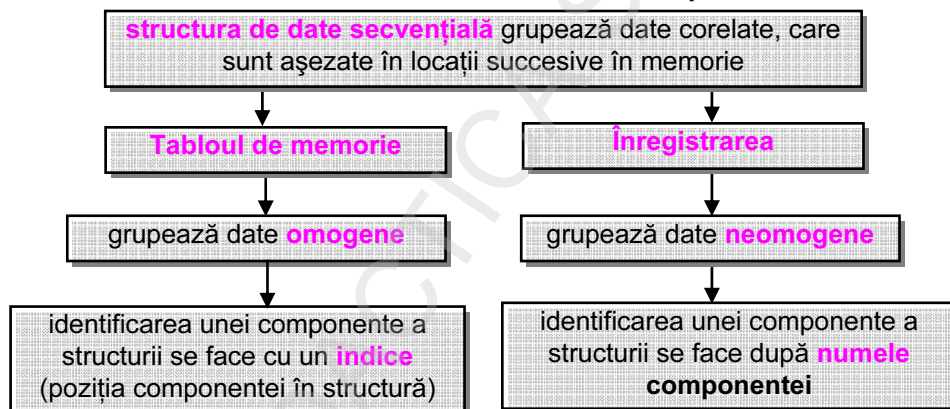
- o structură de date **temporară**;
- o structură de date **cu acces direct** (este permis accesul direct la un câmp);
- o structură de date **statică** (la compilare, i se alocă un anumit spațiu de memorie corespunzător sumei lungimii tuturor câmpurilor, spațiu care nu mai poate fi modificat în timpul executării programului).

Implementarea unei înregistrări se face:

- **Din punct de vedere logic.** Înregistrarea este o structură de date neomogenă, cu elemente care pot fi de tipuri diferite. Localizarea unui element în cadrul structurii se face printr-un nume.
- **Din punct de vedere fizic.** Înregistrării i se alocă o zonă de memorie contiguă, de dimensiune fixă, egală cu suma lungimii tuturor câmpurilor. Dimensiunea alocată unui câmp este determinată de tipul datei memorate în câmp. Localizarea unui câmp se face prin calcularea adresei câmpului față de un element de referință (adresa la care este memorată înregistrarea), ținând cont de dimensiunea câmpurilor precedente.

La fel ca și tabloul de memorie, înregistrarea este o **structură de date secvențială** sau **liniară** (componentele structurii sunt așezate în locații succesive de memorie).

Așadar, **înregistrarea este o zonă continuă de memorie internă căreia i se atribuie un nume și care permite memorarea mai multor date de tipuri diferite. Aceste date pot fi tratate ca un tot unitar sau ca date elementare independente.**



## 2.4.1. Implementarea înregistrării în limbajul C++

### 2.4.1.1. Declararea variabilei de tip înregistrare

În limbajul C++, pentru a putea crea și manipula o structură de date de tip înregistrare, trebuie să definiți un tip de dată numit **structură**.

**Structura** este un tip de dată definită de utilizator care reunește un grup de variabile, sub același nume.

Forma generală a instrucțiunii prin care puteți defini structura este:

```

struct <nume structură>
{<tip dată 1> <nume 11>, <nume 12>, ..., <nume 1n>;
 <tip dată 2> <nume 21>, <nume 22>, ..., <nume 2n>;

 <tip dată m> <nume m1>, <nume m2>, ..., <nume mn>; };
```

unde `<nume_structură>` este numele tipului de dată care reunește mai multe variabile de memorie de tipul `<tip_dată_i>`, identificate prin `<nume_i1>`, `<nume_i2>`, ..., `<nume_in>`. Numele structurii este **opțional**. Variabilele definite în structură se numesc **membrii structurii** și corespund câmpurilor înregistrării. Tipul `<tip_dată_i>` poate fi:

- un tip de bază (de exemplu, `int`, `float`, `char` etc.),
- un tip utilizator definit anterior (cu declarația `typedef` sau `struct`) sau
- un tip utilizator definit chiar în cadrul structurii (tipul tablou de memorie sau tipul înregistrare).

### Observații:

1. Declararea unei structuri se termină cu caracterul `;` deoarece este o instrucțiune.
2. Printr-o instrucțiune de declarare a unei structuri nu se creează o variabilă de memorie, ci un tip de dată utilizator (tipul de dată structurat, sau înregistrare). Pentru a folosi în program o variabilă de memorie de acest tip, ea trebuie declarată:

```
<nume_structură> <nume_variabilă>;
```

Pentru a putea manipula o structură de date de tip înregistrare, trebuie să **definiți**:

1. **un tip de dată structură** prin care descrieți colecția de câmpuri ale înregistrării (numele și tipul lor) și
2. **o variabilă de memorie** care va avea ca tip de dată numele structurii.

Se pot folosi următoarele trei variante de declarare a tipului înregistrare și a variabilei de acest tip:

#### Varianta 1

```
struct <nume_structură> {<descriere_câmpuri_înregistrare>;
<nume_structură> <nume_variabilă>;
```

#### Varianta 2

```
struct <nume_structură>
{<descriere_câmpuri_înregistrare>} <nume_variabilă>;
```

#### Varianta 3

```
struct {<descriere_câmpuri_înregistrare>} <nume_variabilă>;
```

### Observație:

În cea de a treia variantă, nu s-a atribuit un nume structurii definite. Aceasta înseamnă că nu veți putea să definiți ulterior o altă variabilă de memorie cu acest tip.

**Exemplul 1** – Pentru a utiliza o înregistrare în care să memorați datele referitoare la o persoană (numele, prenumele, adresa, localitatea, județul și codul poștal) veți crea mai întâi un tip de dată structură cu numele **adresa** și apoi o variabilă de memorie **adr\_pers** care va avea tipul **adresa**.

```
struct adresa
{char nume[20], pren[20], adr[40], loc[20], jud[20];
 unsigned long cod_p;};
adresa adr_pers;
```

sau

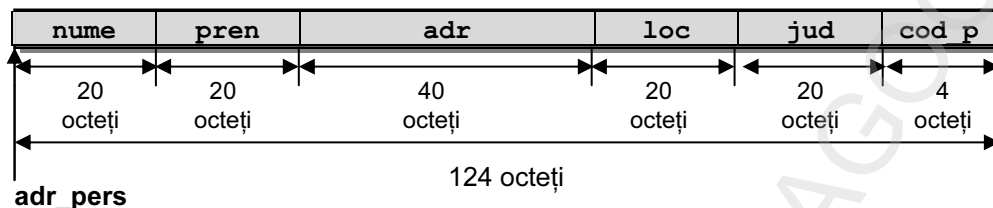
```
struct adresa
{char nume[20], pren[20], adr[40], loc[20], jud[20];
 unsigned long cod_p;} adr_pers;
```

sau

```
struct
```

```
{char nume[20], pren[20], adr[40], loc[20], jud[20];
 unsigned long cod_p;} adr_pers;
```

Compilatorul va rezerva variabilei de memorie **adr\_pers** 124 de octeți.



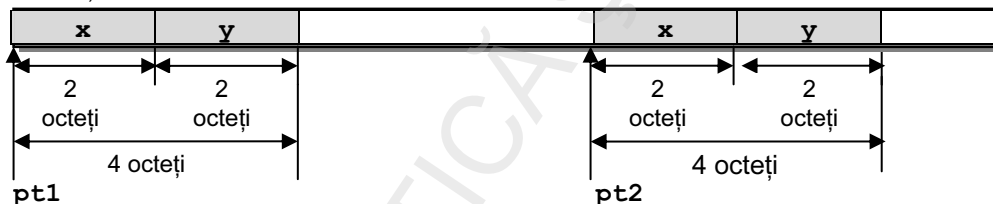
**Exemplul 2** – Pentru a utiliza înregistrări în care să memorați coordonatele unui punct din plan (**x** și **y**), veți crea mai întâi un tip de dată structură cu numele **punct** și apoi două variabile de memorie, **pt1** și **pt2**, care vor avea tipul **punct**.

```
struct punct {int x,y};
punct pt1,pt2;
```

sau

```
struct punct {int x,y;} pt1,pt2;
```

Compilatorul va rezerva pentru fiecare variabilă de memorie de tip **punct** (**pt1** și **pt2**) câte 4 octeți.



### Observație:

Chiar dacă există două câmpuri cu același nume, câmpurile **x**, respectiv câmpurile **y**, ele nu pot fi confundate – deoarece unul aparține înregistrării **pt1**, iar celălalt înregistrării **pt2**.



Creați o înregistrare care să descrie obiectul triunghi (lista de attribute ale acestui obiect) și o înregistrare care să descrie obiectul dată calendaristică.



La fel ca și tabloul de memorie, o structură de date de tip înregistrare poate fi inițializată la declararea ei, atribuindu-se valori câmpurilor:

```
struct punct {int x,y};
punct a={3,4},b={2,5};
```

S-au definit două înregistrări de tip **punct**: **a**, care are coordonatele **x=3** și **y=4**, respectiv **b**, care are coordonatele **x=2** și **y=5**.

### 2.4.1.2. Accesul la câmpurile înregistrării

Pentru prelucrarea datelor dintr-o înregistrare trebuie să aveți acces la fiecare câmp al înregistrării. Pentru a avea acces la un câmp al înregistrării se folosește **operatorul punct** (**.**):

```
<nume_variabilă_înregistrare>.<nume_câmp>
```

**Atenție**

Operatorul punct este **operatorul de selecție a membrului unei structuri** și leagă numele structurii de numele membrului, adică, în expresia **a.b** – **a** trebuie să fie de tip structură, iar **b** trebuie să fie de tip membru al acelei structuri. Rezultatul furnizat de expresie este valoarea membrului selectat. Punctul este un **operator binar** și are **prioritate maximă**.

**Exemplu** – Valorile câmpurilor din înregistrarea **adr\_pers** se pot stabili prin atribuirea unor constante:

```
strcpy(adr_pers.num, "Popescu"); strcpy(adr_pers.pren, "Vlad");
strcpy(adr_pers.adr, "Str. Rozelor nr. 5");
strcpy(adr_pers.loc, "Eforie Sud"); strcpy(adr_pers.jud, "Constanta");
adr_pers.cod_p=123456;
```

sau prin citirea de la tastatură:

```
cout<<"Numele "; cin.get(adr_pers.num,20); cin.get();
cout<<"Prenumele "; cin.get(adr_pers.pren,20); cin.get();
cout<<"Adresa "; cin.get(adr_pers.adr,40); cin.get();
cout<<"Localitatea "; cin.get(adr_pers.loc,20); cin.get();
cout<<"Judetul "; cin.get(adr_pers.jud,20); cin.get();
cout<<"Codul postal "; cin>>adr_pers.cod_p;
```

Prin referirea:

**<nume\_variabilă\_înregistrare>**

se identifică întreaga înregistrare (toate câmpurile înregistrării).

**Observație.** Unei înregistrări (o variabilă de memorie de tip structură) **i se poate atribui valoarea unei alte înregistrări de același tip.**

**Exemplu:**

```
struct punct {int x, y};
punct a,b; a.x=10; a.y=20; b=a;
cout<<b.x<<" "<<b.y; //afișează 10 20
```

**Observație.** Tipul parametrului unui subprogram poate fi de tip înregistrare – deoarece este un tip definit de utilizator. Condiția este ca tipul înregistrare să fie definit ca variabilă globală (tipul de dată utilizator trebuie definit înaintea subprogramului).

**Exemplu:**

```
struct punct {int x, y};
void afiseaza(punct a) {cout<<a.x<<" "<<a.y;}
void main() {punct a={3,4}; afiseaza(a);}
```

## Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi structura de date de tip înregistrare pentru a prelucra datele.

**Enunțul problemei 1:** *Se citesc de la tastatură coordonatele a două puncte din plan – a și b. Să se afișeze distanța dintre cele două puncte.*

```
#include <iostream.h>
#include <math.h>
void main()
{struct punct {int x,y};
```

```
punct a,b;
cout<<"coordonatele punctului a - x: "; cin>>a.x;
cout<<" - y: "; cin>>a.y;
cout<<"coordonatele punctului b - x: "; cin>>b.x;
cout<<" - y: "; cin>>b.y;
cout<<"distanța dintre punctele a și b este ";
cout<<sqrt(pow(a.x-b.x,2)+pow(a.y-b.y,2)); }
```

**Enunțul problemei 2:** Se citește de la tastatură numărătorul și numitorul a două fracții. Să se calculeze suma și produsul celor două fracții, să se simplifice rezultatele obținute și apoi să se afișeze.

În variabilele **f1** și **f2** se memorează cele două fracții, în variabila **s** se calculează suma lor, iar în variabila **p** se calculează produsul lor. Pentru a simplifica fracțiile sumă și produs trebuie calculat, pentru fiecare fracție, cel mai mic divizor comun dintre numărător și numitor. Pentru calculul lui se folosește funcția **cmmdc()**. Pentru calculul sumei celor două fracții se folosește funcția **suma()**, iar pentru produsul lor funcția **produs()**.

```
#include <iostream.h>
struct fractie {int x,y;};
int cmmdc (int a,int b)
{while (a!=b) if (a>b) a-=b; else b-=a; return a;}
fractie suma(fractie f1, fractie f2)
{fractie s; int a;
 s.x=f1.x*f2.y+f1.y*f2.x; s.y=f1.y*f2.y; a=cmmdc(s.x,s.y); s.x/=a; s.y/=a;
 return s;}
fractie produs(fractie f1, fractie f2)
{fractie p; int a;
 p.x=f1.x*f2.x; p.y=f1.y*f2.y; a=cmmdc(p.x,p.y); p.x/=a; p.y/=a;
 return p;}
void main()
{fractie f1,f2,s,p;
 cout<<"prima fractie - numaratorul: "; cin>>f1.x;
 cout<<" - numitor: "; cin>>f1.y;
 cout<<"a doua fractie - numaratorul: "; cin>>f2.x;
 cout<<" - numitor: "; cin>>f2.y;
 s=suma(f1,f2); cout<<"suma este "<<s.x<<"/"<<s.y<<endl;
 p=produs(f1,f2); cout<<"produsul este "<<p.x<<"/"<<p.y<<endl; }
```

### Temă



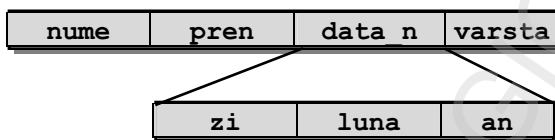
Se citește de la tastatură numărătorul și numitorul a două fracții. Să se compare cele două fracții și să se afișeze fracția care este mai mare.

## 2.4.2. Înregistrări imbricate

În unele cazuri, unul dintre câmpurile înregistrării poate să fie și el, la rândul lui, de tip înregistrare.

**Exemplul 1** – Să considerăm o înregistrare care trebuie să conțină următoarele informații despre o persoană: numele, prenumele, data nașterii și vârsta. Înregistrarea va avea următoarea structură de câmpuri: **nume**, **pren**, **data\_n** și **varsta**. Unul dintre câmpurile

înregistrării (`data_n` folosit pentru data nașterii) este și el tot de tip înregistrare. În acest caz, trebuie definite două înregistrări: una pentru attributele persoanei și alta pentru attributele datei de naștere. Puteți folosi una dintre următoarele variante:



**Varianta 1** – Tipul de dată înregistrare `data` se declară înaintea structurii care caracterizează persoana și care conține câmpul `data_n`, care este de tipul `data`:

```

struct data {unsigned zi, luna, an;};
struct persoana
{char nume[20], pren[20];
 data data_n;
 unsigned varsta;};
persoana pers;

```

Pentru a ne referi la ziua de naștere a unei persoane, se va folosi identificatorul:

`pers.data_n.zi`

**Varianta 2** – Tipul câmpului `data_n` se declară în structura care caracterizează persoana:

```

struct persoana
{char nume[20], pren[20];
 struct {unsigned zi, luna, an;} data_n;
 unsigned varsta;};
persoana pers;

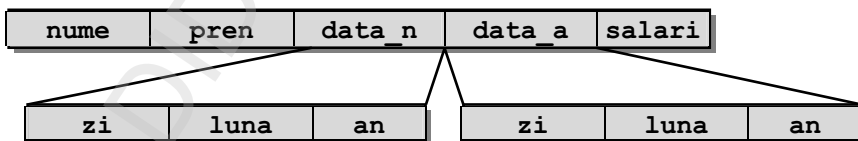
```

Pentru a ne referi la ziua de naștere a unei persoane, se va folosi identificatorul:

`pers.data_n.zi`

**Exemplul 2** – Să considerăm o înregistrare care trebuie să conțină următoarele informații despre o persoană: numele, prenumele, data nașterii, data angajării și salariul. Înregistrarea va avea următoarea structură de câmpuri: `nume`, `pren`, `data_n`, `data_a` și `salariu`.

Două dintre câmpurile înregistrării (`data_n` folosit pentru data nașterii și `data_a` folosit pentru data angajării) sunt tot de tip înregistrare. În acest caz trebuie definite două înregistrări: una pentru attributele persoanei și alta pentru attributele unei date calendaristice care să fie folosită pentru data nașterii și pentru data angajării. Puteți folosi una dintre următoarele variante:



**Varianta 1** – Tipul de dată înregistrare `data` se declară înaintea structurii care caracterizează persoana și care conține câmpul `data_n` care este de tipul `data`:

```

struct data {unsigned zi, luna, an;};
struct persoana
{char nume[20], pren[20];
 data data_n, data_a;
 unsigned long salariu;};
persoana pers;

```

Pentru a calcula diferența dintre anul angajării și anul nașterii persoanei respective, se va calcula expresia:

`pers.data_a.an - pers.data_n.an`



**Varianta 2** – Tipul câmpurilor `data_n` și `data_a` se declară în structura care caracterizează persoana:

```
struct persoana
{char nume[20], pren[20];
 struct {unsigned zi, luna, an;} data n, data a;
 unsigned varsta;};
persoana pers;
```

Pentru a calcula diferența dintre anul angajării și anul nașterii persoanei respective, se va calcula expresia:

`pers.data_a.an - pers.data_n.an`

**Observație:** Dacă o structură de date conține în interiorul său una sau mai multe structuri, se spune că acele structuri sunt **imbricate**. Pentru a identifica un câmp aflat în interiorul unor astfel de structuri imbricate, se construiește identificatorul pornind din exterior către interior, până se ajunge la câmp. Astfel, presupunând că structura `S1` conține structura `S2`, care conține structura `S3`, și așa mai departe, până la structura `Sn`, iar `v1`, `v2`, `v3`, ..., `vn` sunt variabile care au tipul de dată al acestor structuri, identificarea câmpului **câmp** definit în structura `Sn` se va face cu expresia:

`<v1>.<v2>.<v3>...<vn>.<câmp>`

### Atenție

Asociativitatea **operatorului de selecție a membrului unei structuri** este de la stânga la dreapta, pentru că în final trebuie obținută adresa membrului a cărui valoare trebuie furnizată. De aceea este foarte importantă ordinea în care sunt scrise structurile în expresie.

### Temă

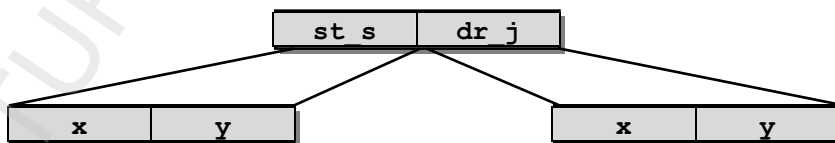
Creați o înregistrare care să conțină următoarele informații despre o persoană: numele, prenumele, adresa, locația domiciliului, locația nașterii, data nașterii și numărul de telefon. Înregistrarea va avea următoarea structură de câmpuri: `nume`, `pren`, `adresa`, `loc_d`, `loc_n`, `data_n` și `telefon`. Câmpul `data_n` este de tip înregistrare, ca în exemplul precedent. Câmpul `adresa` este de tip înregistrare și conține următoarele informații: strada, numărul, blocul, scara, etajul și apartamentul. Câmpurile `loc_d` și `loc_n` sunt de tip înregistrare și conțin următoarele informații: orașul și județul. Câmpul `telefon` este de tip înregistrare și conține următoarele informații: numărul telefonului fix și numărul telefonului mobil.

## Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi structuri imbricate.

**Enunțul problemei 1:** Se citesc de la tastatură coordonatele a două colțuri ale unui dreptunghi: colțul din stânga sus și colțul din dreapta jos. Să se calculeze diagonala dreptunghiului.

Dreptunghiul este caracterizat de două atribute: colțul din stânga sus și colțul din dreapta jos (câmpurile `st_s` și `dr_j`). Fiecare dintre aceste atribute este caracterizat la rândul său de două atribute: coordonatele punctului față de abscisă și ordonată (câmpurile `x` și `y`):



```
#include <iostream.h>
#include <math.h>
```



```
cout<<" - punct 2 - x = "; cin>>d.lat.pt2.x;
cout<<" - y = "; cin>>d.lat.pt2.y;
cout<<"Lungimea - punct 1 - x = "; cin>>d.lung.pt1.x;
cout<<" - y = "; cin>>d.lung.pt1.y;
cout<<" - punct 2 - x = "; cin>>d.lung.pt2.x;
cout<<" - y = "; cin>>d.lung.pt2.y;
l1=sqrt(pow(d.lat.pt1.x-d.lat.pt2.x,2)+
 pow(d.lat.pt1.y-d.lat.pt2.y,2));
l2=sqrt(pow(d.lung.pt1.x-d.lung.pt2.x,2)+
 pow(d.lung.pt1.y-d.lung.pt2.y,2));
cout<<sqrt(pow(l1,2)+ pow(l2,2));
```

Descrierea datelor prelucrate în program se mai poate face și în varianta următoare:

```
struct dreptunghi
{struct segment
 {struct punct {int x,y;} pt1,pt2;} lat,lung;};
dreptunghi d;
```

Înregistrării care caracterizează dreptunghiul *i* se poate adăuga un nou atribut: diagonala dreptunghiului:

```
struct dreptunghi
{struct segment
 {struct punct {int x,y;} pt1,pt2;} lat,lung;
 unsigned diag;};
dreptunghi d;
```

sau

```
struct dreptunghi
{struct {struct {int x,y;} pt1,pt2;} lat,lung;
 unsigned diag;};
dreptunghi d;
```

calculul mării diagonalei dreptunghiului făcându-se astfel:

```
d.diag = sqrt(pow(sqrt(pow(d.lat.pt1.x-d.lat.pt2.x,2)+
 pow(d.lat.pt1.y-d.lat.pt2.y,2)),2)+
 pow(sqrt(pow(d.lung.pt1.x-d.lung.pt2.x,2)+
 pow(d.lung.pt1.y-d.lung.pt2.y,2)),2));
```



### Temă



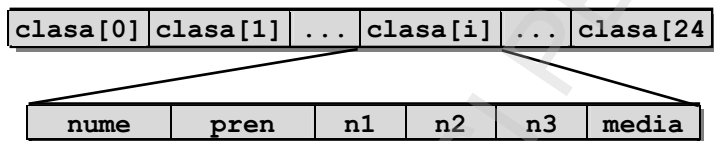
1. Se citesc de la tastatură două intervale de timp exprimate în ore, minute și secunde. Să se calculeze și să se afișeze suma celor două intervale de timp.
2. Se citește de la tastatură o dată calendaristică exprimată în zi, lună și an. Se mai citește și un număr *n*. Să se afișeze data care va fi peste *n* zile. (**Indicație:** se va folosi un vector cu 12 elemente în care se va memora în fiecare element numărul de zile din luna corespunzătoare poziției din vector, prima lună fiind ianuarie.)
3. Se citesc de la tastatură două date calendaristice. Să se afișeze care dată este mai recentă și diferența în zile dintre cele două date.
4. Într-o înregistrare, se păstrează atributele unui triunghi, care sunt cele 3 segmente ce reprezintă laturile triunghiului din plan. Atributele segmentului sunt punctele de la extremitățile lui (precizate prin coordonatele lor). Afișați dacă:
  - a) mărimile celor trei segmente pot fi mărimile laturilor unui triunghi;
  - b) cele trei segmente formează un triunghi.

### 2.4.3. Tablouri de înregistrări

Pentru a caracteriza situația școlară a unui elev la o anumită disciplină, într-un semestru, trebuie să se definească o înregistrare elev, cu următoarea structură de câmpuri:

```
struct elev
{char nume[20], pren[20];
 unsigned n1, n2, n3;
 float media;};
elev a;
```

Pentru a caracteriza situația școlară a tuturor elevilor dintr-o clasă, la o anumită disciplină, într-un semestru, trebuie să se definească un vector `clasa`, ale cărui elemente sunt de tip înregistrare `elev` și care va avea atâtea elemente câți elevi sunt în clasă (de exemplu, 25 de elemente). **Elementele vectorului sunt omogene**, fiind de același tip, adică înregistrări cu aceeași structură:



Declararea acestui vector se face cu instrucțiunea:

```
elev clasa[25];
```

Identificarea unui câmp (de exemplu, câmpul `nume`) care conține numele elevului `i`, din clasă, se face cu expresia:

```
clasa[i].nume;
```

Variabila `e` este de același tip cu elementele vectorului `clasa`, și următoarele operații de atribuire sunt corecte:

```
a = clasa[i]; clasa[i] = a;
clasa[i].nume=e.nume; a.nume = clasa[i].nume;
```

### Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi vectorii cu înregistrări.

**Enunțul problemei:** Într-o clasă sunt maxim 30 de elevi, fiecare elev fiind identificat prin nume și prenume. Elevul poate primi maxim 5 note la o disciplină, pe semestru. Se citesc de la tastatură: numărul de elevi din clasă și, pentru fiecare elev, numele, prenumele și notele. Dacă are mai puțin de 5 note, notelor lipsă li se va atribui valoarea 0, iar în calculul mediei se vor lua numai notele diferite de 0. Să se calculeze și să se afișeze mediile elevilor din clasă la acea disciplină.

Se folosesc variabilele `s` și `k` pentru a calcula suma notelor unui elev, respectiv numărul de note diferite de zero.

```
#include <iostream.h>
#include <math.h>
void main()
{struct elev {char nume[20], pren[20];
 unsigned n1, n2, n3, n4, n5;
 float media;};
```

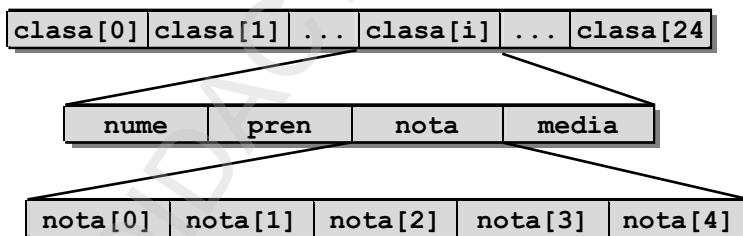
```

elev clasa[30]; int n,i,s,k;
cout<<"nr. de elevi din clasa "; cin>>n;
//se citesc elementele vectorului (câmpurile înregistrării
//pentru fiecare elev)
for (i=0;i<n;i++)
{cin.get(); cout<<"elevul "<<i+1<<endl;
cout<<"nume "; cin.get(clasa[i].nume,20); cin.get();
cout<<"prenume "; cin.get(clasa[i].pren,20); cin.get();
cout<<"nota 1 "; cin>>clasa[i].n1;
cout<<"nota 2 "; cin>>clasa[i].n2;
cout<<"nota 3 "; cin>>clasa[i].n3;
cout<<"nota 4 "; cin>>clasa[i].n4;
cout<<"nota 5 "; cin>>clasa[i].n5;}
for (i=0;i<n;i++) //se calculează media pentru fiecare elev
{s=0,k=0;
if (clasa[i].n1!=0) {s+=clasa[i].n1;k++;}
if (clasa[i].n2!=0) {s+=clasa[i].n2;k++;}
if (clasa[i].n3!=0) {s+=clasa[i].n3;k++;}
if (clasa[i].n4!=0) {s+=clasa[i].n4;k++;}
if (clasa[i].n5!=0) {s+=clasa[i].n5;k++;}
clasa[i].media=(float)s/k;}
for (i=0;i<n;i++) //se afișează media fiecărui elev
{cout<<clasa[i].nume<<" "<<clasa[i].pren<<" ";
cout<<clasa[i].media<<endl;} }

```

### Observație:

Pentru a simplifica algoritmul de prelucrare, notele fiecărui elev pot fi și ele grupate într-un vector, structura vectorului `clasa` devenind:



iar programul:

```

#include <iostream.h>
#include <math.h>
void main()
{struct elev {char nume[20],pren[20];
 unsigned nota[5];
 float media;};
elev clasa[30]; int n,i,j,s,k;
cout<<"nr. de elevi din clasa "; cin>>n;
for (i=0;i<n;i++)
{cin.get(); cout<<"elevul "<<i+1<<endl;
cout<<"nume "; cin.get(clasa[i].nume,20); cin.get();

```

```

cout<<"prenume "; cin.get(clasa[i].pren,20); cin.get();
for (j=0;j<5;j++) {cout<<"nota "<<j+1<<" ";
 cin>>clasa[i].nota[j];}

for (i=0;i<n;i++)
{ s=0,k=0;
 for (j=0;j<5;j++)
 if (clasa[i].nota[j]!=0) {s+=clasa[i].nota[j]; k++;}
 clasa[i].media=(float)s/k;}
for (i=0;i<n;i++)
{cout<<clasa[i].nume<<" "<<clasa[i].pren<<" ";
 cout<<clasa[i].media<<endl;} }

```



### Temă



1. Într-un vector cu înregistrări, se păstrează atributele a  $n$  dreptunghiuri: lungimea, lăţimea, aria şi perimetrul. Numărul  $n$  şi dimensiunile laturilor dreptunghiurilor se introduc de la tastatură. Să se afişeze dreptunghiul cu suprafaţa cea mai mare şi dreptunghiul cu perimetrul cel mai mic.
2. Într-un vector cu înregistrări, se păstrează atributele a  $n$  dreptunghiuri: lungimea, lăţimea şi diagonală. Numărul  $n$  şi dimensiunile laturilor dreptunghiurilor se introduc de la tastatură. Se mai citeşte de la tastatură o valoare  $d$ . Să se afişeze dreptunghiurile a căror diagonală are dimensiunea  $d$ .
3. Într-un vector cu înregistrări se păstrează atributele a  $n$  puncte. Atributele punctului sunt coordonatele şi cadranul în care se găseşte. Numărul  $n$  şi coordonatele punctelor se introduc de la tastatură. Să se afişeze punctele grupate după cadran.

## Evaluare

### Răspundeţi:

1. Scrieţi câte o înregistrare pentru fiecare dintre următoarele obiecte specificate prin lista lor de atribute:
  - a) carte = (titlu, autor, an\_aparitie, editura, nr\_pagini, pret);
  - b) paralelipiped dreptunghic = (x1,y1,z1; x2,y2,z2; x3,y3,z3 – coordonatele celor trei colţuri reprezentative);
  - c) maşină = (marcă, număr, culoare, serie, an fabricaţie, preţ);
  - d) calculator = (marcă, memorie, procesor, imprimantă, monitor);
  - e) elev = (număr\_matricol, nume, prenume, anul\_naşterii, clasa);
  - f) material = (cod, denumire, unitate\_măsură, preţ, cantitate\_intrată, cantitate\_ieşită, stoc).

2. Se consideră următoarele declaraţii:

```

struct s1 {char a; int b;};
struct s2 {long a; float b;};
struct s3 {s1 a; s2 b;};
s3 a,b;

```

Precizaţi tipul câmpurilor: **a.a.a**; **a.a.b**; **a.b.a**; **a.b.b**; **b.b.b**; **b.b.a**; **b.a.a** şi **b.a.b**.

3. Ce se va afişa în urma execuţiei următoarei secvenţe de program:

```

typedef int mat[4][4];
struct ex {char a[20]; int b; char c; mat d;};
ex a[5][5]; int i,j,k,l;

```

```

for (i=0; i<5; i++)
 for (j=0; j<5; j++)
 for (k=0; k<4; k++)
 for (l=0; l<4; l++)
 a[i][j].d[k][l]=i+j+k+l;
cout<<a[4][4].d[3][3]<<endl<<a[i-1][j-1].d[k-1][l-1];

```

4. Comparați următoarele programe. Ce realizează? Analizați modul în care subprogramele transmit rezultatele către modulul apelant. Scrieți a treia variantă de program folosind variabile globale.

```

struct polinom
{int n;
 int c[20];};
void citire(polinom &p)
{int i;
 cout<<"gradul"; cin>>p.n;
 for (i=0; i<=p.n; i++)
 {cout<<"coeficientul lui x^"
 <<p.n-i<<" = ";
 cin>>p.c[i];} }
int valoare (polinom p, int a)
{int x=0,i;
 for (i=0; i<=p.n ; i++)
 x=x*a+p.c[i];
 return x;}
void main()
{polinom p; int a;
 citire(p); cout<<"a= "; cin>>a;
 cout<<valoare(p,a);}

```

```

struct polinom
{int n;
 int c[20];};
polinom citire()
{int i; polinom p;
 cout<<"gradul = "; cin>>p.n;
 for (i=0; i<=p.n; i++)
 {cout<<"coeficientul lui x^"
 <<p.n-i<<" = ";
 cin>>p.c[i];}
 return p;}
int valoare (polinom p, int a)
{int x=0,i;
 for (i=0; i<=p.n; i++) x=x*a+p.c[i];
 return x;}
void main()
{polinom p; int a;
 p=citire(); cout<<"a= "; cin>>a;
 cout<<valoare(p,a);}

```

### Adevărat sau Fals:

1. Următoarea declarație este corectă:

```

struct s1
{struct s2 {int a; float b;}; int a;};

```

2. Următoarele declarații nu sunt corecte:

```

struct s2 {int a; float b;};
struct s1 {s2 a; int b;};

```

3. Următoarele declarații sunt corecte:

```

struct s1 {s2 a; int b;};
struct s2 {int x; float y;};

```

### Alegeți:

1. Care dintre următoarele variante reprezintă o declarare corectă a unei variabile structurate cu două componente, una de tip întreg și una de tip real?

a. `int float x[2];`

b. `float x[2];`

c. `struct  
{float x;  
int a;} x;`

d. `typedef struct  
{long a;  
float b;} inreg;  
inreg x;`





- e. citirea datelor dintr-un fișier text – pe primul rând al fișierului va fi scris numărul de obiecte care se prelucrează ( $n$ ) și apoi, pe următoarele  $n$  rânduri, separate prin spațiu, pentru fiecare obiect, valorile atributelor care sunt date de intrare.
1. Într-un vector cu înregistrări se păstrează atributele a  $n$  segmente din plan. Atributele segmentului sunt punctele de la extremitățile lui (precizate prin coordonatele lor) și lungimea segmentului. Să se afișeze segmentul (prin coordonatele punctelor din extremități) care are lungimea cea mai mare și segmentul care are lungimea cea mai mică. Dacă există mai multe segmente care îndeplinesc aceste proprietăți, să se afișeze toate segmentele.
  2. Într-o clasă sunt maxim 30 de elevi, fiecare elev fiind identificat prin nume și prenume. Elevul poate primi maxim 5 note la o disciplină, pe semestru, și o notă la teză. Se citesc de la tastatură: numărul de elevi din clasă și, pentru fiecare elev, numele, prenumele și notele. Dacă are mai puțin de 5 note, notelor lipsă li se va atribui valoarea 0. Să se calculeze și să se afișeze mediile elevilor din clasă la acea disciplină, în ordinea:  
a) descrescătoare a mediilor;      b) alfabetică a numelui și prenumelui.
  3. În doi vectori cu înregistrări (`clasa1` și `clasa2`) sunt păstrate informații despre elevii din două clase: numele, prenumele și media generală. Cei doi vectori sunt ordonați crescător după medie. Să se interclaseze cei doi vectori în vectorul `clase`, care trebuie să conțină suplimentar și informația despre clasa elevului. Să se afișeze informațiile din acest vector. Să se afișeze informații despre elevul cu media cea mai mică și despre elevul cu media cea mai mare. Să se afișeze media mediilor generale pe fiecare clasă și pe ambele clase împreună. Să se afișeze ce clasă are media pe clasă mai mare.
  4. Într-un vector cu înregistrări se păstrează atributele a  $n$  triunghiuri: laturile, aria, perimetrul și tipul triunghiului (oarecare, isoscel etc.). Să se afișeze informațiile despre triunghiuri, grupate după tip (grupa triunghiurilor isoscele, a triunghiurilor oarecare etc.).
  5. Într-un vector cu înregistrări, se păstrează atributele a  $n$  numere: valoarea numărului și descompunerea lui în factori primi. Se introduc de la tastatură două numere naturale  $p$  și  $k$ . Să se afișeze numerele care conțin în descompunerea lor factorul prim exprimat prin  $p$  la puterea  $k$ . (**Indicație:** atributul *descompunerea numărului în factori primi* va fi implementat ca un vector de înregistrări care conține un câmp pentru divizorul prim și un câmp pentru puterea divizorului; la înregistrarea corespunzătoare numărului va trebui adăugat câmpul *număr de factori primi*).
  6. Într-o fabrică sunt  $m$  secții, iar în fiecare secție lucrează  $n$  muncitori. Atributele fiecărui muncitor sunt: numele, prenumele, secția, salariul orar, timpul lucrat (exprimat în ore) și salariul. Pentru fiecare muncitor se va calcula salariul (care este egal cu produsul dintre salariul orar și timpul lucrat). Să se afișeze totalul salariului pe fiecare secție și pe întreaga fabrică.
  7. O persoană este caracterizată de următoarele atribute: sex, vârstă, înălțime, greutate. Se analizează un grup de 100 de persoane. Să se afișeze:  
a. procentul de femei și procentul de bărbați;  
b. vârsta medie a grupului, vârsta medie a femeilor și vârsta medie a bărbaților.  
c. procentul de femei și procentul de bărbați peste vârsta medie a grupului;
  8. Pentru grupul de 100 de persoane analizat la problema anterioară, să se afișeze:  
a. înălțimea medie a grupului și înălțimea medie a femeilor;  
b. procentul de femei și procentul de bărbați peste înălțimea medie a grupului;  
c. greutatea medie a grupului, a femeilor și a bărbaților;  
d. procentul de femei și procentul de bărbați peste greutatea medie a grupului.

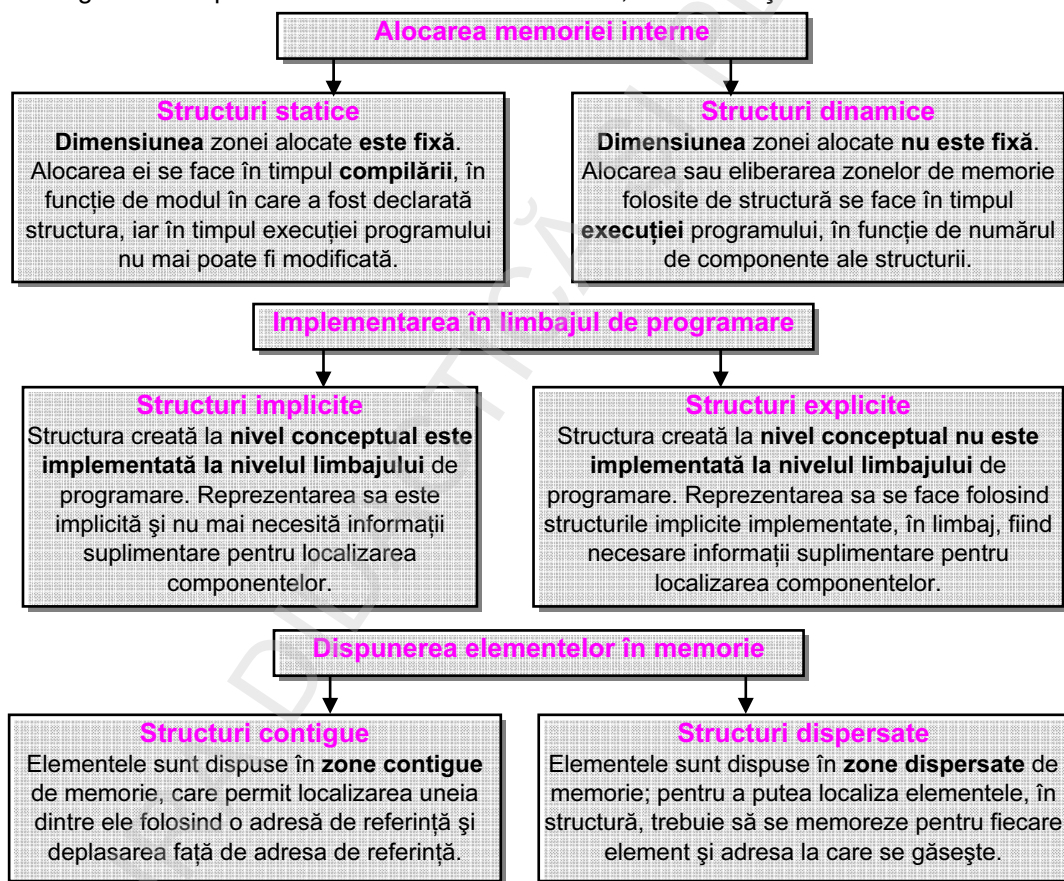
## 2.5. Lista

Pentru rezolvarea unei probleme, în multe cazuri se pot folosi mai mulți algoritmi, din care trebuie ales cel mai eficient:

- la nivel **conceptual**, alegerea algoritmului se face în funcție de complexitatea algoritmilor descoperiți pentru rezolvarea problemei, fiind preferat algoritmul care are **complexitate mai mică** – algoritmul care consumă cel mai puțin resursele calculatorului;
- la nivel **logic**, alegerea se face în funcție de modul în care pot fi implementați algoritmi în limbajul de programare ales, fiind preferat algoritmul care are o **implementare mai ușoară** – algoritmul care consumă cel mai puțin resursele programatorului.

Nu întotdeauna cei doi algoritmi coincid, programatorul urmând să decidă criteriul folosit pentru alegerea algoritmului. Un rol important în implementarea eficientă a unui algoritm îl joacă **modul în care au fost organizate datele în colecția de date**.

Pe lângă criteriile pentru clasificarea datelor studiate, mai există și următoarele **criterii**:



Structurile statice au un mare dezavantaj, deoarece alocarea zonei de memorie se face la compilarea programului (în funcție de modul în care a fost declarată structura), iar în timpul execuției programului pot să apară următoarele cazuri:

- spațiul alocat structurii este **insuficient**;
- spațiul alocat structurii este **mult mai mare decât este necesar**.

Acest dezavantaj este eliminat prin folosirea structurilor dinamice, la care dimensiunea zonei de memorie alocate nu mai este fixă, deoarece alocarea memoriei se face în timpul execuției programului, în funcție de numărul de componente ale structurii la acel moment.

### Studiu de caz

**Scop:** exemplificarea modului în care identificați problemele în care puteți folosi structura de date de tip listă pentru a le rezolva.

**Enunțul problemelor** pentru care trebuie aleasă o structură de date și conceput algoritmul pentru prelucrarea lor:

1. *Într-o bibliotecă, există o colecție de cărți organizate în ordinea alfabetică a autorilor. Un cititor poate împrumuta o carte (se extrage o carte din colecție) sau poate înapoia o carte (se inserează o carte în colecție). Toate aceste operații trebuie executate astfel încât să se păstreze organizarea în ordine alfabetică a autorilor.*
2. *La o benzinărie s-a format o coadă de așteptare. Mașinile sunt servite în ordinea venirii: prima mașină sosită este servită, iar ultima mașină venită se așază la sfârșitul cozii. Toate aceste operații trebuie executate astfel încât să se păstreze ordinea în care au sosit mașinile și s-au așezat la coadă.*
3. *Într-o stivă de cărți, volumele sunt așezate în ordinea alfabetică a titlurilor. Trebuie extrasă o carte din stivă fără a deranja modul în care sunt ordonate cărțile în stivă.*

La nivel **conceptual**, toate aceste colecții de date reprezintă un șir de date de același tip, care trebuie prelucrate prin inserarea și extragerea de elemente, **păstrându-se o anumită ordine de aranjare a elementelor**. Dacă la nivel **logic** s-ar alege soluția de a grupa aceste elemente într-o structură de date de tip **vector**, algoritmi de prelucrare – care sunt algoritmi de actualizare a vectorilor – vor necesita multe deplasări de elemente care consumă timp de prelucrare.

Caracteristicile operațiilor de prelucrare a vectorilor sunt:

- Vectorul este o structură de date care are lungimea fizică fixă, iar implementarea lui la nivel fizic este un proces prin care se face conversia locației conceptuale a unui element, în locația reală, din memoria internă.
- Orice operație de adăugare sau extragere a unui element din colecție modifică lungimea logică a vectorului.
- Pentru a insera un element în colecție, trebuie deplasate toate elementele spre dreapta, începând cu poziția inserării.
- Pentru a extrage un element din colecție, trebuie deplasate toate elementele spre stânga, de la sfârșit, până în poziția din care se extrage.

În cazul structurilor care trebuie să-și păstreze în timpul exploatării ordonarea după un anumit criteriu, mecanismul vectorilor este greoi. În aceste cazuri, se poate alege ca soluție de implementare a structurii de date **lista** – care **nu este o structură fizică de organizare a datelor**, ci o structură logică, ce degreuează programatorul de ordonarea după indice a structurii, impusă de vectori. Listele sunt structuri de date care, spre deosebire de vectori, au marele avantaj că permit și implementare prin alocarea dinamică a memoriei.



**Lista** este o structură de date **logică, liniară**, cu date **omogene**, în care fiecare element are un succesor și un predecesor, exceptând primul element, care nu are decât succesor, și ultimul element, care nu are decât predecesor.

Aparent, vectorul și lista par să fie aceeași structură de date: sunt structuri **omogene** și **secvențiale (liniare)** în care fiecare element are un succesor și un predecesor, cu excepția primului și a ultimului element, care au numai succesor, respectiv numai predecesor. Între cele două structuri există însă următoarele **deosebiri**:

| Criteriul                                | Tabloul de memorie                                                                                                                                                                                                                                                                                 | Lista                                                                                                                                                                                                                                                                                                                                 |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Disponerea elementelor în memorie</b> | Sunt structuri <b>contigue</b> .<br>Predecesorul, elementul și succesorul se găsesc în locații contigue prin mecanismul de implementare a structurii de date la nivel fizic.                                                                                                                       | Pot fi structuri <b>dispersate</b> . Predecesorul, elementul și succesorul nu sunt în locații de memorie contigue. Programatorul trebuie să definească mecanismul prin care elementele structurii vor fi legate unele de altele pentru a putea fi regăsite în memorie.                                                                |
| <b>Implementarea în limbaj</b>           | Sunt structuri <b>implicite</b> . Fiind o structură fizică, este implementată în limbajul de programare. Nu necesită informații suplimentare pentru localizarea elementelor structurii în memoria internă, deoarece mecanismul prin care este implementată fizic asigură identificarea elementelor | Sunt structuri <b>explicite</b> . Fiind o structură logică, trebuie aleasă o metodă de implementare folosind structurile fizice existente. Necesită informații suplimentare pentru localizarea elementelor structurii în memoria internă, deoarece mecanismul prin care este implementată fizic nu asigură identificarea elementelor. |
| <b>Alocarea memoriei</b>                 | Sunt structuri <b>statice</b> .                                                                                                                                                                                                                                                                    | Sunt structuri <b>statice</b> sau <b>dinamice</b> , în funcție de implementarea aleasă.                                                                                                                                                                                                                                               |

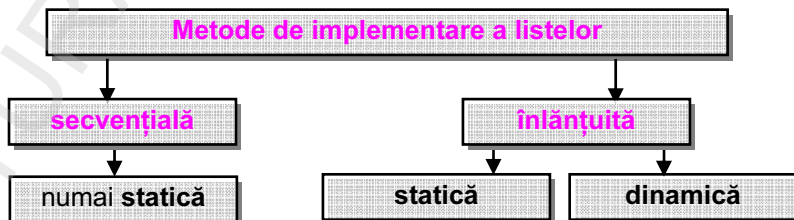
Se definește **lungimea listei** ( $n$ ) ca fiind numărul de elemente ale listei. **Lista vidă** este lista care are lungimea 0 (nu are nici un element). Elementele listei se mai numesc și **noduri**.

### 2.5.1. Implementarea listelor în limbajul C++

În funcție de modul în care se alocă memoria internă, se pot folosi metodele:



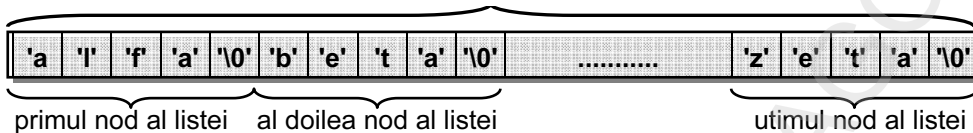
În funcție de modul în care sunt aranjate elementele în listă, se pot folosi metodele:



### 2.5.1.1. Implementarea prin alocare secvențială

Nodurile listei sunt stocate într-un **bloc contiguu de locații de memorie cu adrese consecutive**. De exemplu, dacă avem o listă formată din cuvinte de maximum 4 caractere, acestea vor fi scrise într-un bloc contiguu de locații de 5 octeți.

**bloc contiguu de locații de memorare de 5 octeți**



Această implementare este cea mai simplă – și se face folosind un vector de șiruri de caractere:

```
typedef char nod[5];
nod lista[100];
```

S-a declarat o listă care poate avea maxim 100 de noduri. Informația dintr-un nod este de tip șir de caractere cu lungimea de 5 caractere. Acest tip de listă se mai numește și **listă contiguu**.

În implementarea prin alocare secvențială, algoritmi pentru operațiile de prelucrare a listei sunt cei folosiți pentru operațiile de prelucrare a vectorilor și necesită foarte multe operații de deplasare a elementelor în vector.

### 2.5.1.2. Implementarea prin alocare înlănțuită

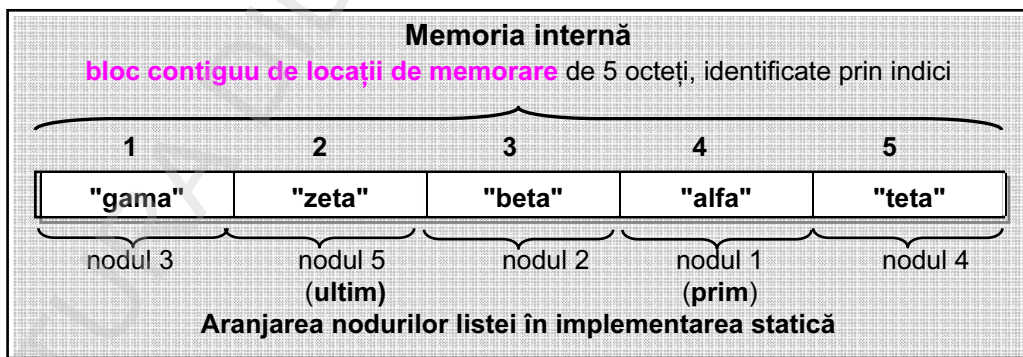
Nodurile listei **nu mai sunt stocate succesiv** în memorie.

**Exemplu** – O listă este formată din 5 cuvinte (noduri), fiecare cuvânt având maxim 4 caractere. Nodurile listei se exploatează în ordine alfabetică:

Lista = {"alfa", "beta", "gama", "teta", "zeta"}

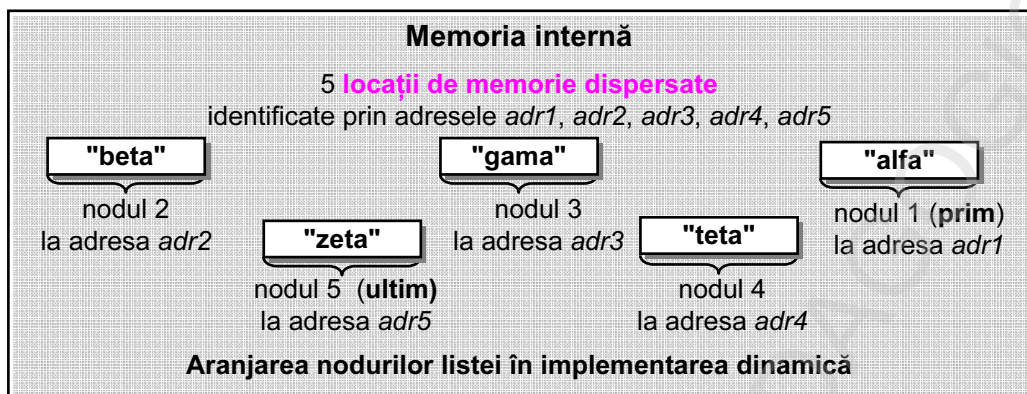
Această implementare se poate face atât static, cât și dinamic, între cele două implementări existând următoarele diferențe:

→ în implementarea **statică**, nodurilor listei li se alocă un **bloc contiguu de locații de memorie** (zona de memorie alocată vectorului);



→ în implementarea **dinamică**, nodurile listei ocupă **locații dispersate din memorie** (a căror adresă poate fi păstrată cu ajutorul pointerilor).





Deoarece în ambele cazuri de implementare nodurile nu sunt aranjate succesiv, ci aleatoriu, în memorie, trebuie implementat un mecanism prin care să se precizeze ordinea reală a acestor noduri (ordinea în care se înlănțuiesc în listă). Aceasta înseamnă că:

- trebuie cunoscut locul din care începe lista (lanțul de noduri), adică poziția primului nod din listă (nodul **prim**) – în exemplu, nodul "*alfa*";
- trebuie cunoscut locul în care se termină lista, adică poziția ultimului nod din listă (nodul **ultim**) – în exemplu, nodul "*zeta*";
- pentru fiecare nod din listă, cu excepția ultimului nod, trebuie cunoscut nodul care este succesorul lui – de exemplu, pentru nodul "*gama*" trebuie cunoscut că succesorul său este nodul "*teta*".

Metoda folosită pentru implementare este ca un nod al listei să conțină două tipuri de informații: **informația propriu-zisă** și **informația**

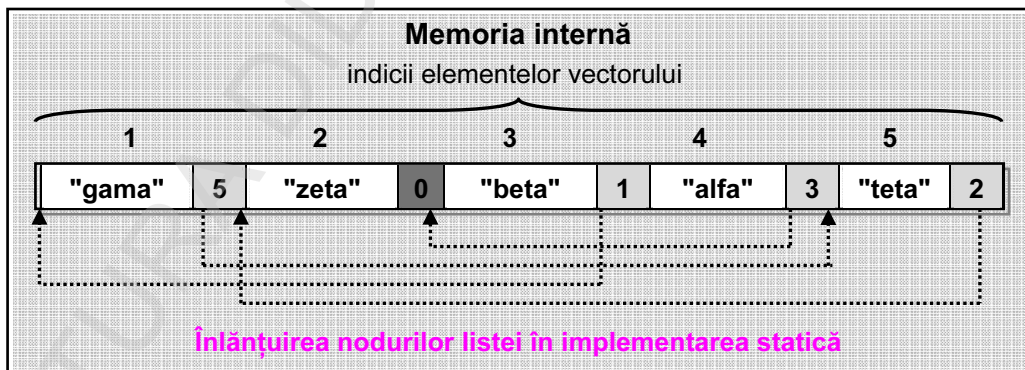
**Informația propriu-zisă**

**Informația pentru legătură**

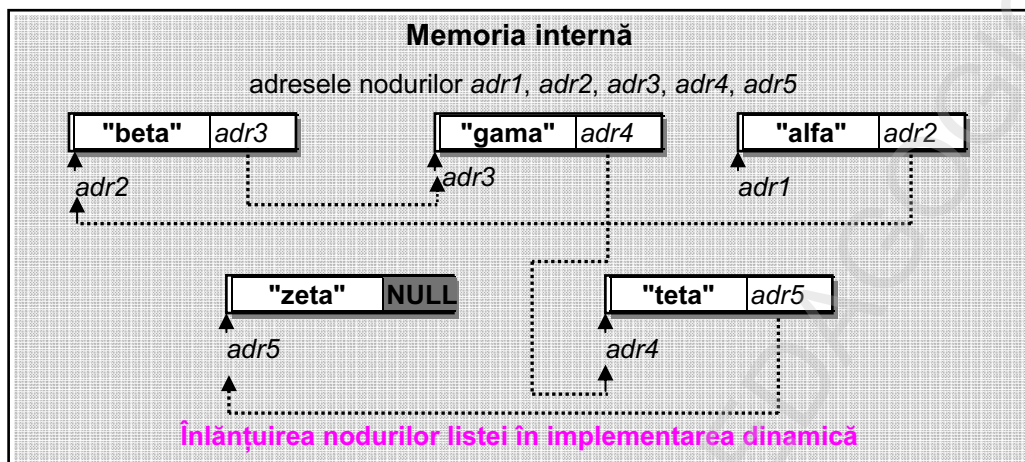
**de legătură** (informația prin care se identifică nodul care urmează în structură, adică adresa acestui nod). Informația pentru legătură va fi:

- în cazul implementării **statice** – **indicele din vector al succesorului**;
- în cazul implementării **dinamice** – **adresa succesorului** exprimată printr-un pointer.

În ambele cazuri, informația de legătură memorată în ultimul nod este constanta **NULL** sau 0 (care semnifică faptul că ultimul nod nu se leagă de nimic).







Implementarea statică pentru alocarea înlanțuită se face folosind un vector de înregistrări:

```
typedef unsigned adresa;
struct nod
{
 char sir[5]; //informația propriu-zisă
 adresa urm; //informația pentru legătură
}
nod lista[100];
```

În variabila **urm** se va memora adresa succesului (indicele din vector al succesului). În general, putem spune că, pentru implementarea statică a listei și pentru prelucrarea cu algoritmi specifici unei liste, trebuie declarate următoarele tipuri de date și variabile:

```
const unsigned NMAX=100;
typedef unsigned adresa;
struct nod
{
 <tip 1> <info 11>, <info 12>, ..., <info 1n>;
 <tip 2> <info 21>, <info 22>, ..., <info 2n>;

 <tip_m> <info_m1>, <info_m2>, ..., <info_mn>;
 adresa urm;
}
nod lista[NMAX+1];
```

Câmpurile **<info\_ij>** sunt câmpurile cu informații, iar câmpul **urm** este câmpul care conține informația de legătură (adresa succesului). Se definește constanta **NMAX** pentru a putea folosi în expresii lungimea fizică a vectorului. Deoarece indicele 0 se folosește pentru adresa NULL, în vectorul care implementează lista, elementul cu indicele 0 nu se folosește.

În liste, **algoritmii** de inserare și eliminare a unor noduri din structură se simplifică foarte mult:

- **Inserarea unui nod** constă în alocarea zonei de memorie în care se scrie nodul și legarea lui la structură (stabilirea legăturii cu predecesorul și cu succesul lui).
- **Eliminarea unui nod** constă în ruperea nodului din structură, prin legarea predecesorului său cu succesul lui, și eliberarea zonei de memorie pe care a ocupat-o.

Implementarea structurilor de date cu ajutorul listelor are următoarele avantaje:

- **alocarea dinamică a memoriei** – care o gestionează mult mai eficient;
- **algoritmii de prelucrare** – care sunt mult mai eficienți (se execută mult mai puține operații pentru inserarea și ștergerea unui element).

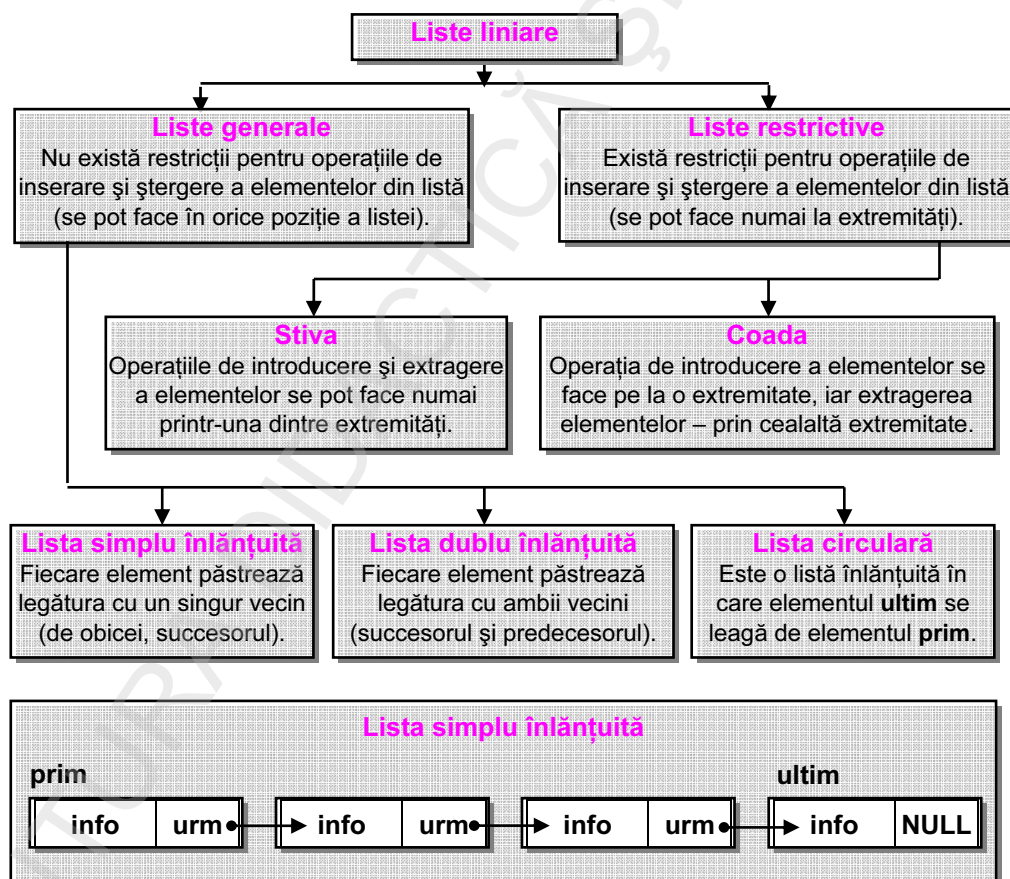
| Metoda de implementare | Avantajul alocării dinamice a memoriei | Avantajul algoritmilor de prelucrare |
|------------------------|----------------------------------------|--------------------------------------|
| statică secvențială    | nu                                     | nu                                   |
| statică înlănțuită     | nu                                     | da                                   |
| dinamică înlănțuită    | da                                     | da                                   |

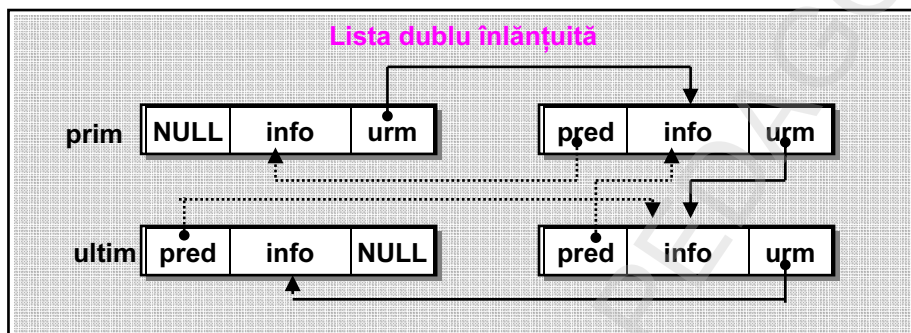
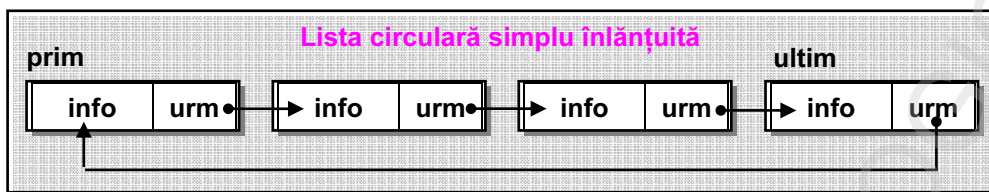
Se observă că implementarea statică secvențială nu aduce niciunul dintre avantajele listelor, fiind o implementare în care lista este de fapt un vector.

**Algoritmii pentru prelucrarea unei liste înlănțuite** sunt aceiași, atât în cazul implementării statice, cât și în cazul implementării dinamice. Deoarece informațiile pe care le aveți nu vă permit încă abordarea alocării dinamice a memoriei, pentru prezentarea algoritmilor se va folosi implementarea statică.

Așadar, lista este o **structură logică de date**, parcursă **liniar**, care are două extremități (**început** și **sfârșit**), în care fiecărui element  $i$  se asociază o **informație suplimentară** referitoare la locul elementului următor, din punct de vedere logic.

## 2.5.2. Clasificarea listelor





### Temă



Reprezentați grafic o listă simplu înlănțuită prin predecesori și o listă circulară dublu înlănțuită.

## 2.5.3. Algoritmi pentru prelucrarea listelor generale

Operațiile care se pot executa asupra listelor sunt:

- **inițializarea listei** – se creează lista vidă;
- **crearea listei** – se adaugă repetat elemente la listă, pornind de la lista vidă;
- **inserarea unui element în listă** – la început, la sfârșit, în interior;
- **eliminarea unui element din listă** – la început, la sfârșit, în interior;
- **parcurgerea listei** – se vizitează elementele listei pentru a obține informații;
- **căutarea în listă a unui element** care îndeplinește anumite condiții;
- **concatenarea** a două liste;
- **divizarea** în două liste.

Pentru a simplifica algoritmi pentru prelucrarea listei, vom considera că un nod al listei conține numai un câmp cu informație și câmpul pentru realizarea legăturii.

```
const unsigned NMAX=100;
typedef unsigned adresa;
struct nod
{int info; adresa urm;};
nod lista[NMAX+1];
adresa prim,ultim,p;
unsigned nr_el,liber[NMAX];
int n;
```

Veți mai folosi următoarele date și structuri de date:

- Variabilele **prim** și **ultim** reprezintă adresele (indicele poziției din vector) în care sunt memorate primul și respectiv ultimul nod; ele vă ajută să identificați extremitățile listei.
- Variabila **p** este adresa (indicele) unui nod curent din listă (este folosită pentru parcurgerea listei).

- Variabila **n** conține valoarea care se atribuie câmpului cu informații.
- Variabila **nr\_el** este lungimea listei (numărul de noduri ale listei).
- Vectorul **liber** este o hartă a nodurilor libere: **liber[p]** are valoarea 1 dacă nodul cu adresa (indicele) **p** este liber și valoarea 0 dacă nodul cu adresa **p** este ocupat. Prin acest vector, se implementează mecanismul de alocare și de eliberare a zonei de memorie necesare unui nod. Atunci când se inserează un nod, mai întâi trebuie să i **se aloce memorie** internă. Căutând în vectorul **liber**, se va putea identifica prima locație liberă din vectorul **lista**. Atunci când se șterge un element, **se eliberează memoria** alocată în vectorul **lista**, prin actualizarea informației din vectorul **liber**.

Există următoarele cazuri speciale de liste:

- **Lista vidă**: atât nodul **prim** cât și nodul **ultim** nu există, și indicii folosiți pentru adresarea lor au valoarea **NULL**:

```
prim = ultim = NULL;
```

- **Lista cu un singur nod**: nodul **prim** este și nodul **ultim** și nu mai este legat de nici un alt element:

```
lista[prim].info = val;
lista[prim].urm = NULL;
ultim = prim;
```

Există următoarele stări ale listei, care trebuie cunoscute în algoritmi de prelucrare:

- **Lista vidă**. Această stare trebuie cunoscută, atunci când se elimină noduri din listă, deoarece în lista vidă nu există noduri care să fie eliminate. Pentru testarea unei liste dacă este vidă, se poate implementa funcția operand **este\_vida()**, care va furniza valoarea 1 („adevărat”), dacă lista este vidă, și valoarea 0 („fals”) dacă lista nu este vidă.

```
int este_vida(adresa prim)
{return prim==NULL;}
```

- **Lista plină**. Această stare poate să apară din cauza implementării statice a listei. Ea trebuie cunoscută atunci când se adaugă noduri la listă, deoarece se poate depăși lungimea fizică a vectorului. În lista plină nu mai există locații libere în vector, în care să se adauge noduri. Pentru testarea unei liste dacă este vidă, se poate implementa funcția operand **este\_plina()**, care va furniza valoarea 1 („adevărat”), dacă lista este plină, și valoarea 0 („fals”) dacă lista nu este plină.

```
int este_plina()
{return nr_el == NMAX;}
```

### 2.5.3.1. Inițializarea listei

Prin acest algoritm **se creează lista vidă**, în care atât nodul **prim** cât și nodul **ultim** nu există, și vor avea valoarea **NULL**. În vectorul **liber**, toate elementele au valoarea 1 (toate elementele din vectorul listei sunt libere).

**Implementarea algoritmului.** Se folosește funcția procedurală **init()** ai cărei parametri **prim** și **ultim** de tip **adresa** se transmit prin referință, deoarece sunt parametri de ieșire.

```
void init(adresa &prim, adresa &ultim)
{ultim=prim=NULL; nr_el=0;
 for (adresa p=1;p<=NMAX; p++) liber[p]=1;}
```

### 2.5.3.2. Alocarea memoriei

Pentru a putea adăuga un nod la listă – trebuie mai întâi să i **se aloce o zonă de memorie**. Prin acest algoritm se găsește prima poziție liberă în vectorul **lista**.

**Implementarea algoritmului.** Se folosește funcția operand `alloc_mem()` care furnizează adresa (indicele) primei poziții libere în vector – variabila locală `p` de tip adresă.

```
adresa alloc_mem()
{adresa p;
 for (adresa p=1; !liber[p]; p++);
 liber[p]=0; nr_el++;
 return p;}
```

### 2.5.3.3. Crearea listei

Deoarece în algoritmul de prelucrare trebuie să se cunoască adresa primului nod, este importantă **adăugarea primului nod la lista vidă**. Pașii **algoritmului de creare** a unei liste sunt:

**PAS1.** Se adaugă primul nod la listă (nodul **prim**).

**PAS2.** **Cât timp** mai există informație **execută**: se adaugă un nod la listă.

### 2.5.3.4. Adăugarea primului nod la listă

Prin acest algoritm se adaugă primul nod în listă. Spațiul alocat acestui nod este prima poziție din vector (poziția cu indicele 0). În vectorul `liber`, poziția 0 se va înregistra ca ocupată prin atribuirea valorii 0. Nodul **prim** și nodul **ultim** au valoarea 1. Nodul **prim**, fiind unicul nod, nu mai este legat de alt nod.



În lista cu un singur nod, adresa de legătură a nodului **prim** are valoarea **NULL** și atât nodul **prim** cât și nodul **ultim** au aceeași adresă. Pașii executați în acest algoritm sunt:

**PAS1.** Se cere alocarea de memorie pentru nodul **prim**.

**PAS2.** Se scrie informația în nodul **prim**.

**PAS3.** Adresei de legătură a nodului **prim** i se atribuie valoarea **NULL**.

**PAS4.** Nodului **ultim** i se atribuie adresa nodului **prim**.

**Implementarea algoritmului.** Se folosește funcția procedurală `adauga_primul_nod()` ai cărei parametri se transmit astfel: **prim** și **ultim** de tip **adresa** prin referință, deoarece sunt parametri de ieșire, iar **n**, care conține informația utilă, prin valoare, deoarece este parametru de intrare.

```
void adauga_primul_nod(adresa &prim, adresa &ultim, int n)
{prim=alloc_mem(); lista[prim].info=n; lista[prim].urm=NULL; ultim=prim;}
```

### 2.5.3.5. Adăugarea unui nod la listă

Pentru adăugarea unui nod la listă (prin scriere la adresa `p` care i s-a alocat), în funcție de cerințele problemei, se poate folosi unul dintre algoritmul următor:

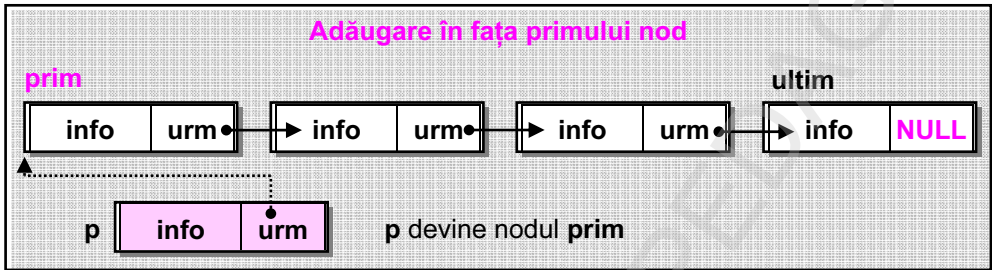
1. **adăugarea în fața primului nod;**
2. **adăugarea după ultimul nod;**
3. **adăugarea într-o poziție în interiorul listei.**

În toate cele trei variante adăugarea se poate face numai dacă lista nu este plină.

### Adăugare în fața primului nod

Pașii executați în acest algoritm sunt:

- PAS1.** Se cere alocarea de memorie pentru nodul **p**.
- PAS2.** Se scrie informația în nodul **p**.
- PAS3.** Dacă lista este vidă, nodul **p** adăugat va fi și nodul **ultim**.
- PAS4.** Nodul **p** se leagă de nodul **prim**.
- PAS5.** Nodul **p** inserat devine nodul **prim**.



**Implementarea algoritmului.** Se folosește funcția procedurală `adauga_inainte_prim()` ai cărei parametri se transmit astfel: **prim** și **ultim** de tip **adresa** prin referință, deoarece sunt parametri de intrare-ieșire și **n** – care conține informația utilă – prin valoare, deoarece este parametru de intrare.

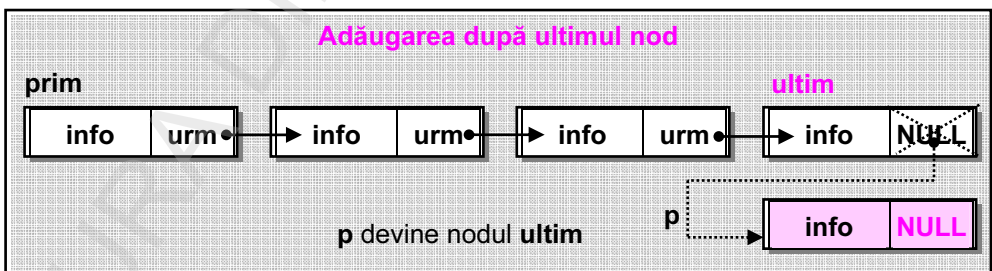
```
void adaug_inainte_prim(adresa &prim, adresa &ultim, int n)
{adresa p; p=alloc_mem();
 lista[p].info=n; lista[p].urm=prim; //nodul p se leagă de nodul prim
 if (este_vidă(prim)) ultim=p;
 prim=p;}
```

//nodul p devine nodul prim

### Adăugare după ultimul nod

Pașii executați în acest algoritm sunt:

- PAS1.** Se cere alocarea de memorie pentru nodul **p**.
- PAS2.** Se scrie informația în nodul **p**.
- PAS3.** Nodul **p** este nod terminal (nu se leagă de nimic – adresa de legătură este NULL).
- PAS4.** Nodul **ultim** se leagă de nodul **p** adăugat.
- PAS5.** Dacă lista este vidă, nodul **p** inserat va fi și nodul **prim**.
- PAS6.** Nodul **p** adăugat devine nodul **ultim**.



**Implementarea algoritmului.** Se folosește funcția procedurală `adauga_dupa_ultim()` ai cărei parametri se transmit astfel: **prim** și **ultim** de tip **adresa** prin referință, deoarece sunt parametri de intrare-ieșire și **n** – care conține informația utilă – prin valoare, deoarece este parametru de intrare.

```

void adaug_dupa_ultim(adresa &prim, adresa &ultim, int n)
{adresa p; p=alloc_mem();
 lista[p].info=n; lista[p].urm=NULL; //nodul p nu se leagă de nimic
 lista[ultim].urm=p; //nodul ultim se leagă de nodul p
 if (este_vidă) prim=p;
 ultim=p;} //nodul p devine nodul ultim

```

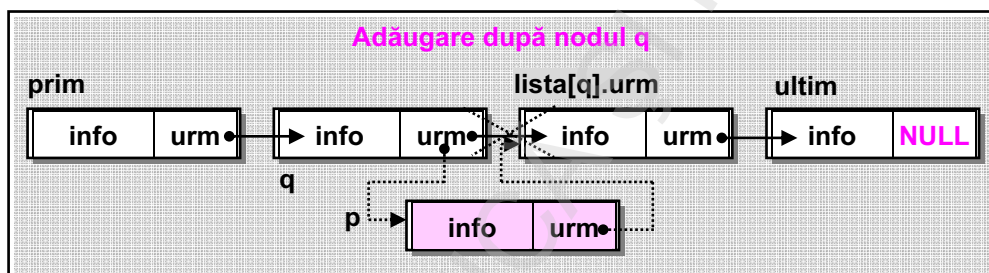
**Adăugarea în interiorul listei** se poate face în două moduri:

- după nodul din poziția  $q$ ;
- înainte de nodul din poziția  $q$ .

### Adăugarea în interiorul listei după nodul din poziția $q$

Pașii algoritmului sunt:

- PAS1.** Se cere alocarea de memorie pentru nodul  $p$ .  
**PAS2.** Se scrie informația în nodul  $p$ .  
**PAS3.** Nodul  $p$  se leagă de succesorul nodului  $q$ .  
**PAS4.** Nodul  $q$  se leagă de nodul  $p$  care se adaugă.  
**PAS5.** Dacă nodul  $q$  a fost ultimul nod, nodul  $p$  adăugat devine nodul  $ultim$ .



**Implementarea algoritmului.** Se folosește funcția procedurală `adauga_dupa()` ai cărei parametri se transmit astfel:  $q$  (adresa nodului după care se face adăugarea) prin valoare, deoarece este parametru de intrare,  $ultim$  (adresa ultimului nod), prin referință, deoarece este parametru de intrare-ieșire, și  $n$  – care conține informația utilă – prin valoare, deoarece este parametru de intrare.

```

void adauga_dupa(adresa q, adresa &ultim, int n)
{adresa p; p=alloc_mem();
 lista[p].info=n; lista[p].urm=lista[q].urm; lista[q].urm=p;
 if (lista[p].urm=NULL) ultim=p;}

```

### Adăugarea în interiorul listei înainte de nodul din poziția $q$

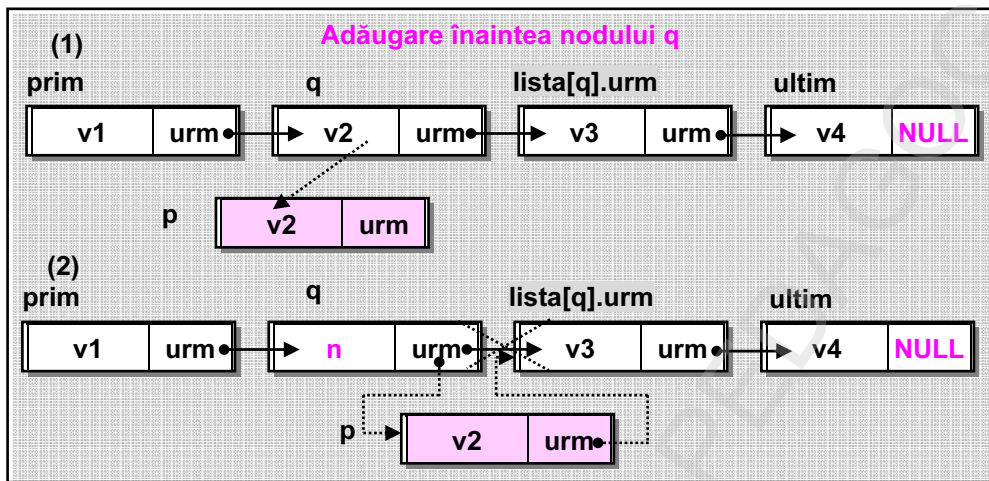
Pentru a insera nodul  $p$  înaintea nodului  $q$ , trebuie să legăm predecesorul nodului  $q$  de nodul  $p$ . Dar, predecesorul unui nod nu este cunoscut. Inserarea unui nod în listă înseamnă, de fapt, inserarea în listă a informației pe care o conține, între alte două informații, și anume: informația din predecesorul nodului  $q$  trebuie să fie anterioară ei, iar informația din nodul  $q$  trebuie să o urmeze. Astfel, în listă nu se va insera nodul  $p$  înainte de nodul  $q$ , ci după el, interschimbând apoi informațiile între cele două noduri. Pașii executați în acest algoritm sunt:

- PAS1.** Se cere alocarea de memorie pentru nodul  $p$ .  
**PAS2.** Se copiază informația din nodul  $q$  în nodul  $p$ .  
**PAS3.** Se scrie în nodul  $q$  informația care trebuie adăugată la listă.  
**PAS4.** Nodul  $p$  se leagă de succesorul nodului  $q$ .



**PAS5.** Nodul  $q$  se leagă de nodul  $p$  adăugat.

**PAS6.** Dacă nodul  $q$  a fost ultimul nod, nodul  $p$  adăugat devine nodul **ultim**.



**Implementarea algoritmului.** Se folosește funcția procedurală **adauga\_in\_fata()** ai cărei parametri se transmit astfel:  $q$  (adresa nodului înaintea căruia se face adăugarea), prin valoare, deoarece este parametru de intrare, **ultim** (adresa ultimului nod), prin referință, deoarece este parametru de intrare-ieșire, și  $n$  – care conține informația utilă – prin valoare, deoarece este parametru de intrare.

```
void adauga_in_fata(adresa q, adresa &ultim, int n)
{
 adresa p; p=alloc_mem();
 lista[p].info=lista[q].info; lista[q].info=n;
 lista[p].urm=lista[q].urm; lista[q].urm=p;
 if (lista[p].urm==NULL) ultim=p;
}
```

### 2.5.3.6. Parcurgerea listei

Prin acest algoritm se vizitează fiecare nod al listei, pornind de la primul nod, până la ultimul nod, în ordinea de înlănțuire a nodurilor – furnizată de câmpul **urm** din nodul vizitat. Lista se parcurge pentru a prelucra informația stocată în noduri.

**Implementarea algoritmului.** Se folosește funcția procedurală **parcurge()** al cărei parametru **prim** de tip **adresa** se transmite prin valoare, deoarece este parametru de intrare.

```
void parcurge(adresa prim)
{
 for (adresa p=prim; p!=NULL; p=lista[p].urm)
 //se prelucreează lista[p].info;
}
```

### 2.5.3.7. Căutarea unui nod în listă

Într-o listă se poate căuta:

1. **Nodul care îndeplinește o anumită condiție**, pe care o notăm cu **conditie** și care este exprimată printr-o expresie logică ce conține cel puțin un câmp cu informație din nod; valoarea ei este dependentă de informația stocată în nod. **Algoritmul** este: se parcurge lista începând de la primul nod, în ordinea de înlănțuire a nodurilor, până se găsește nodul care îndeplinește condiția sau până s-a ajuns la sfârșitul listei.
2. **Nodul care se găsește într-o anumită poziție în listă**, pe care o notăm cu **poz**. **Algoritmul** este: se parcurge lista începând de la primul nod, în ordinea de înlănțuire a nodurilor, până când s-au parcurs **poz** noduri sau până s-a ajuns la sfârșitul listei.

**Implementarea algoritmului.** Se folosește o funcție operând cu tipul **adresa** care va returna adresa nodului găsit. În cazul în care nodul căutat nu există în listă, funcția va returna valoarea **NULL**. În ambele variante:

- parametrul funcției este **prim** de tip **adresa** și se transmite prin valoare, deoarece este parametru de intrare;
- variabila locală **p** de tip **adresa** se folosește pentru parcurgerea listei – este inițializată cu adresa primului nod;
- adresa nodului găsit se memorează în variabila **p** care va fi returnată de funcție.

#### Varianta 1

```
adresa caut(adresa prim)
{for (adresa p=prim; p!=NULL && !conditie; p=lista[p].urm);
 return p;}
```

**Varianta 2** – Variabila locală **nr** de tip **int** se folosește pentru a număra pozițiile parcurse – este inițializată cu valoarea 1.

```
adresa caut(adresa prim, int poz)
{adresa p=prim; int nr=1;
 if (poz>nr el) return NULL;
 else for (;nr<poz; p=lista[p].urm,nr++);
 return p;}
```

### 2.5.3.8. Căutarea succesorului și a predecesorului unui nod

Într-o listă se pot căuta predecesorul și succesorul unui nod care are adresa **p**. Valoarea lui **p** poate să fie furnizată de un algoritm de căutare în listă. Presupunem că lista are cel puțin trei noduri și că există nodul căutat. **Algoritmul** este: se inițializează predecesorul (**pred**) cu adresa primului nod din listă și succesorul (**succ**) cu adresa celui de al treilea nod din listă și se parcurge lista, în ordinea de înălțuire a nodurilor, de la al doilea nod, până la nodul căutat, folosind variabila **q** de tip **adresa**, actualizând de fiecare dată valorile adreselor succesorului și predecesorului.

**Implementarea algoritmului.** Se folosește o funcție procedurală **pred\_succ()** ai cărei parametri se transmit astfel: **prim** (adresa primului nod) și **p** (adresa nodului căruia i se caută succesorul și predecesorul), prin valoare, deoarece sunt parametri de intrare, iar **pred** (adresa predecesorului) și **succ** (adresa succesorului) prin referință, deoarece sunt parametri de ieșire.

```
void pred_succ(adresa prim, adresa p, adresa &pred, adresa &succ)
{adresa q=lista[prim].urm;
 pred=prim; succ=lista[lista[prim].urm].urm;
 while (q!=p) {pred=q; q=lista[q].urm; succ=lista[q].urm;}
```

**Observație.** Prin găsirea predecesorului unui nod, se poate simplifica algoritmul pentru ștergerea ultimului nod.

### 2.5.3.9. Eliminarea unui nod din listă

Eliminarea unui nod se face numai dacă lista nu este vidă. După eliminarea din listă a nodului de la adresa **p**, se va elibera zona de memorie ocupată de nod.

Se folosește funcția procedurală **elibereaza()** al cărei parametru **p** de tip **adresa** se transmite prin valoare (este parametru de intrare).

```
void elibereaza(adresa p)
{liber[p]=1; nr_el--;}
```

Pentru eliminarea unui nod din listă, în funcție de cerințele problemei, se poate folosi unul dintre algoritmii următori:

1. **eliminarea primului nod;**
2. **eliminarea ultimului nod;**
3. **eliminarea unui nod din interiorul listei.**

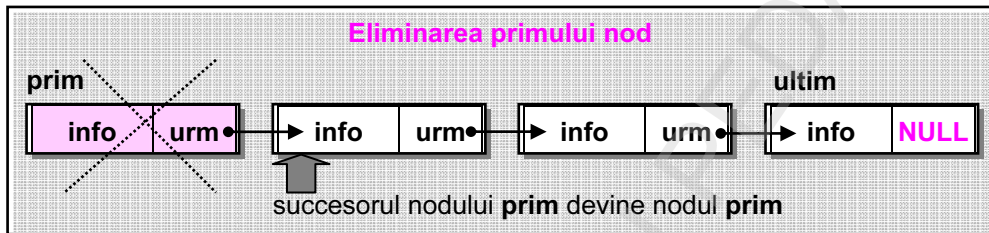
### Eliminarea primului nod

Pașii executați în acest algoritm sunt:

**PAS1.** Se salvează adresa nodului **prim** în variabila de tip adresă **q**.

**PAS2.** Succesorul nodului **prim** devine nodul **prim**.

**PAS3.** Se eliberează zona de memorie de la adresa memorată în variabila **q**.



**Implementarea algoritmului.** Se folosește funcția procedurală **elimina\_prim()** al cărei parametru **prim** de tip **adresa** se transmite prin referință, deoarece este parametru de intrare-ieșire.

```
void elimina_prim(adresa &prim)
{adresa q=prim; prim=lista[prim].urm; elibereaza(q);}
```

### Eliminarea ultimului nod

Pașii executați în acest algoritm sunt:

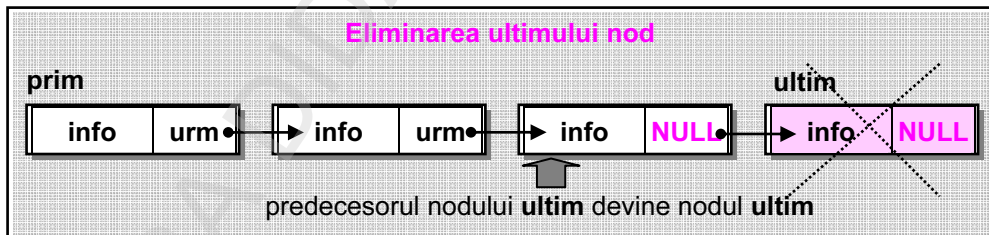
**PAS1.** Se salvează adresa nodului **ultim** în variabila **q** de tip **adresa**.

**PAS2.** Se caută predecesorul ultimului nod, prin parcurgerea listei începând de la primul nod, până la predecesorul nodului terminal (nodul care nu se leagă de nimic – adresa de legătură are valoarea NULL).

**PAS3.** Predecesorul nodului **ultim** devine nod terminal (adresa de legătură este NULL).

**PAS4.** Predecesorul nodului **ultim** devine nodul **ultim**.

**PAS5.** Se eliberează zona de memorie de la adresa memorată în variabila **q**.



**Implementarea algoritmului.** Se folosește funcția procedurală **elimina\_ultim()** al cărei parametru **ultim** de tip **nod** se transmite prin referință deoarece este parametru de intrare-ieșire.

```
void elimina_ultim(adresa prim, adresa &ultim)
{adresa p,q=ultim;
 for (p=prim; lista[lista[p].urm].urm!=NULL; p=lista[p].urm);
 //lista[p].urm este adresa ultimului nod
 //și nodul p este predecesorul ultimului nod
 lista[p].urm=NULL; ultim=p; elibereaza(q);}
```

### Eliminarea unui nod din interiorul listei

Pentru a elimina nodul **p** aflat în interiorul listei, trebuie să legăm predecesorul nodului **p** de succesul lui. Dar, predecesorul unui nod nu este cunoscut, ci numai succesul lui. Eliminarea unui nod din listă înseamnă de fapt eliminarea din listă a informației pe care o conține. Astfel, din listă nu se va elimina nodul **p**, ci succesul său (care este cunoscut), după ce informația din el a fost copiată în nodul **p**. Pașii executați în acest algoritm sunt:

**PAS1.** Se salvează adresa succesului nodului **p** în variabila **q** de tip **adresa**.

**PAS2.** Se copiază în nodul **p** toată informația din succesul lui (informația propriu-zisă și informația de legătură).

**PAS3.** Se cere eliberarea zonei de memorie de la adresa memorată în variabila **q**.

**PAS4.** Dacă succesul nodului **p** era nodul **ultim**, atunci nodul **p** devine nodul **ultim**.

**Implementarea algoritmului.** Se folosește funcția procedurală **elimina()** ai cărei parametri sunt de tip **nod**: **p** (adresa nodului care se elimină), care se transmite prin valoare, deoarece este parametru de intrare, și **ultim**, care se transmite prin referință, deoarece este parametru de intrare-ieșire.

```
void elimina(adresa p, adresa &ultim)
{adresa q=lista[p].urm; //nodul q este succesul nodului p
 lista[p].info=lista[q].info; lista[p].urm=lista[q].urm;
 elibereaza(q);
 if (lista[p].urm)==NULL) ultim=p;}
```



În vectorul din figură este implementată o listă liniară înlănțuită ale cărei noduri conțin caractere. Stabiliți corect legăturile între elementele listei și precizați care sunt valorile pentru **prim** și **ultim** în cazul în care nodurile listei:

a) sunt ordonate alfabetic;

b) formează cuvântul *carte*.

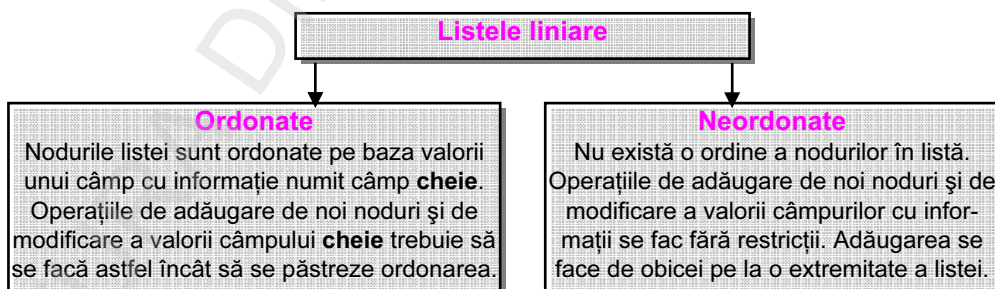
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| r |   | a |   | e |
|   |   |   | c |   |
|   |   |   |   | t |

Eliminați din listă caracterul *r*. Adăugați la listă caracterul *i*. Stabiliți corect legăturile între elementele listei și precizați care sunt valorile pentru **prim** și **ultim** – în cazul în care nodurile listei:

a) sunt ordonate alfabetic;

b) formează cuvântul *caiet*.

### 2.5.3.10. Prelucrarea listelor



În cazul listelor ordonate, dacă informația din nodul listei este o dată elementară, cheia de ordonare va fi data elementară. Dacă informația din nodul listei este o înregistrare, cheia de ordonare va fi unul dintre câmpurile înregistrării.

**Atenție**

În general, rezolvarea problemelor în care organizați datele sub formă de liste presupune folosirea algoritmilor prezentați, astfel:

1. **Se creează lista** prin folosirea următorilor algoritmi:
  - Se creează lista vidă – **algoritmul pentru crearea listei vide**.
  - Se adaugă câte un nod la listă. Poziția în care se adaugă nodul depinde de cerințele problemei. Dacă lista nu este ordonată, adăugarea se poate face la sfârșitul sau la începutul listei – **algoritmul pentru adăugare la începutul sau la sfârșitul listei**. Dacă lista este ordonată, se parcurge mai întâi lista pentru a găsi poziția de inserare – **algoritmul pentru căutarea unui nod în listă** – după care se inserează noul nod în poziția găsită – **algoritmul pentru adăugare în interiorul listei**.
2. **Se întreține lista** prin folosirea următorilor algoritmi:
  - Se adaugă noi noduri la listă. Ca și la crearea listei, în funcție de cerințele problemei se va folosi unul dintre **algoritmii de adăugare**.
  - Se elimină noduri din listă. Mai întâi se parcurge lista pentru a găsi nodul care se elimină – **algoritmul pentru căutarea unui nod în listă** – după care se elimină nodul din poziția găsită folosind unul dintre **algoritmii pentru eliminare** – în funcție de poziția în care a fost găsit nodul.
  - Se modifică informația dintr-un nod al listei. Mai întâi se parcurge lista pentru a găsi nodul în care se va modifica informația – **algoritmul pentru căutarea unui nod în listă**. Dacă lista nu este ordonată sau dacă informația care se modifică nu face parte dintr-un câmp cheie dintr-o listă ordonată, se modifică valoarea câmpului respectiv. Dacă lista este ordonată și, prin modificarea valorii câmpului, se poate distruge această ordonare, se modifică valoarea câmpului, se salvează nodul listei într-o variabilă intermediară, se elimină din vechea poziție – **algoritmul pentru eliminarea unui nod din listă** –, se parcurge lista pentru a găsi noua poziție – **algoritmul pentru căutarea unui nod în listă** – și se inserează nodul în poziția găsită – **algoritmul pentru adăugarea unui nod la listă**.
3. **Se obțin informații din listă**. Se parcurge lista – **algoritmul de parcurgere a listei** – și se vizitează fiecare nod al listei pentru a extrage din el informațiile necesare.

**Studiu de caz**

**Scop:** exemplificarea modului în care, pentru rezolvarea problemei, folosiți algoritmii de prelucrare a listelor simplu înlănțuite și implementarea lor cu ajutorul subprogrameelor.

**Enunțul problemei 1:** *Se citesc, într-o listă, cifrele unui număr care are o valoare foarte mare. Citirea se va face până când numărul citit nu este o cifră. Să se afișeze inversul acestui număr.*

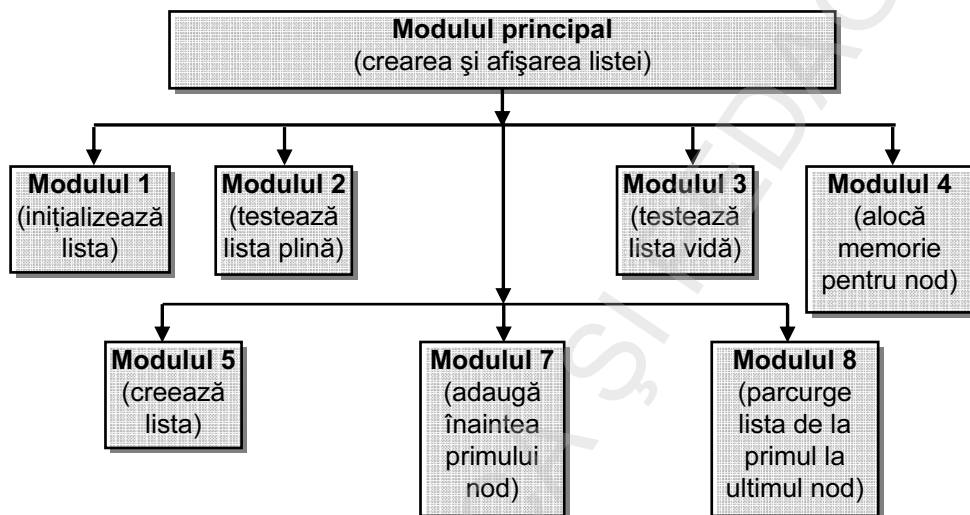
Nodurile listei nu sunt ordonate conform unui criteriu. Cifrele numărului se vor citi de la tastatură și se vor scrie în listă prin adăugare în fața primului nod. Pentru a afișa inversul numărului se va parcurge lista de la primul nod până la ultimul, și se va afișa informația din fiecare nod. Deoarece în acești algoritmi nu este necesară informația despre adresa ultimului nod, din subprograme au fost eliminate prelucrările care se referă la acesta.

**Folosind tehnica top-down, problema poate fi împărțită în subprobleme**, iar algoritmul de rezolvare a unei subprobleme este implementat cu ajutorul unui subprogram.

**P1** Se creează lista vidă – subprogramul `init()`.

- P2** Se creează lista cu cifre – subprogramul **create()** – astfel: se citește primul număr și **cât timp** numărul citit este cifră și lista nu este plină – subprogramul **este\_plina()** – **execută**: se adaugă un nod cu cifra respectivă înaintea primului nod din listă – subprogramul **adaug\_inainte\_prim()** – și se citește un număr.
- P3** Dacă lista nu este vidă – subprogramul **este\_vida()** –, **atunci** se parcurge lista de la primul nod până la sfârșitul ei și se afișează informația din fiecare nod – subprogramul **afiseaza()**.

Structura modulelor este următoarea:



Specificațiile subprogramelor sunt următoarele:

- Modulul 1 (init):**
  - **Date de intrare:** niciuna;
  - **Date de ieșire:** **prim** – adresa primului nod al listei;
  - **Funcția** modulului: crearea listei vide.
- Modulul 2 (este\_plina):**
  - **Date de intrare:** niciuna;
  - **Date de ieșire:** valoarea de adevărat sau fals a stării de listă plină;
  - **Funcția** modulului: obținerea informației despre starea de listă plină.
- Modulul 3 (este\_vida):**
  - **Date de intrare:** **prim** – adresa primului nod al listei;
  - **Date de ieșire:** valoarea de adevărat sau fals a stării de listă vidă;
  - **Funcția** modulului: obținerea informației despre starea de listă vidă.
- Modulul 4 (aloc\_mem):**
  - **Date de intrare:** niciuna;
  - **Date de ieșire:** **p** – adresa alocată pentru nod;
  - **Funcția** modulului: alocarea unei zone de memorie (un element din vector) pentru un nod nou.
- Modulul 5 (create):**
  - **Date de intrare:** **prim** – adresa primului nod al listei;
  - **Date de ieșire:** **prim** – adresa primului nod al listei;
  - **Funcția** modulului: se creează lista prin citirea cifrelor de la tastatură.

**6. Modulul 6 (adaug\_inainte\_prim):**

- **Date de intrare:** **prim** și **n** – adresa primului nod al listei și cifra din nodul care se adaugă;
- **Date de ieșire:** **prim** – adresa primului nod al listei;
- **Funcția** modului: se adaugă la listă un nod cu informație înaintea primului nod.

**7. Modulul 7 (afișează):**

- **Date de intrare:** **prim** – adresa primului nod al listei;
- **Date de ieșire:** niciuna;
- **Funcția** modului: vizitarea nodurilor listei de la primul până la ultimul, pentru a afișa informația din nod.

Programul obținut este:

```
#include <iostream.h>
const unsigned NMAX=100;
typedef unsigned adresa;
struct nod {unsigned cifra;
 adresa urm;};
nod lista[NMAX+1];
int nr_el, liber[NMAX];

int este_plina()
{return nr_el == NMAX;}

int este_vida(adresa prim)
{return prim==NULL;}

void init(adresa &prim)
{prim=NULL; nr_el=0;
 for (adresa p=1;p<=NMAX;p++) liber[p]=1;}

adresa aloc_mem()
{adresa p;
 for (p=1; !liber[p]; p++);
 liber[p]=0; nr_el++;
 return p;}

void adaug_inainte_prim(adresa &prim, int n)
{adresa p=aloc_mem();
 lista[p].cifra=n; lista[p].urm=prim; prim=p;}

void creare(adresa &prim)
{unsigned cif;
 cout<<"Scrieti cifrele numarului:"<<endl;
 cout<<"cifra= "; cin>>cif;
 while (cif<=9 && !este_plina()) //se adauga noduri la lista
 {adaug_inainte_prim(prim,cif);
 cout<<"cifra= "; cin>>cif;}}

void afiseaza(adresa prim)
{for (adresa p=prim; p!=NULL; p=lista[p].urm) cout<<lista[p].cifra;}

void main()
{adresa prim; init(prim); creare(prim);}
```



```
if (!este_vida(prim)) {cout<<"Numarul invers este: "; afiseaza(prim);}
else cout<<"Nu exista numar";}
```

### Temă



1. Refaceți programul folosind prototipurile funcțiilor.
2. Scrieți următoarele programe în care numerele din nodurile listei se citesc dintr-un fișier text în care sunt scrise pe același rând, separate prin spațiu:
  - a. Se creează o listă cu aceste numere și se afișează în ordinea inversă citirii din fișier numai numerele pare.
  - b. Se creează o listă în care ordinea de acces este inversă celei în care sunt citite numerele. Să se elimine din listă cel mai mic număr și cel mai mare număr.

**Enunțul problemei 2:** Se introduc de la tastatură mai multe numere naturale, până când se introduce numărul 0. Să se afișeze mai întâi numerele pare și apoi numerele impare.

Vom folosi pentru memorarea numerelor o listă înlănțuită, în care vom scrie mai întâi numerele pare și apoi numerele impare. Numerele pare vor fi scrise în listă prin adăugarea în fața primului nod, iar numerele impare prin adăugare după ultimul nod. Problema poate fi descompusă în subprobleme, care vor fi rezolvate cu ajutorul subprogramelor.

**P1** Se creează lista vidă – subprogramul `init()`.

**P2** Se creează lista cu numere – subprogramul `creare()` – astfel: se citește primul număr și se adaugă primul nod la listă – subprogramul `adaug_prim()` –, se citește următorul număr și **cât timp** numărul citit este diferit de zero și lista nu este plină – subprogramul `este_plina()` – **execută: dacă** numărul este par, **atunci** se adaugă înaintea primului nod – subprogramul `adaug_inainte_prim()`; **altfel**, se adaugă după ultimul nod – subprogramul `adaug_dupa_ultim()` – și se citește un nou număr.

**P3** Dacă lista nu este vidă – subprogramul `este_vida()` –, **atunci** se parcurge lista de la primul nod până la sfârșitul ei și se afișează informația din fiecare nod – subprogramul `afiseaza()`.

Programul obținut este:

```
#include <iostream.h>
const unsigned NMAX=100;
typedef unsigned adresa;
struct nod {unsigned nr;
 adresa urm;};
nod lista[NMAX+1];
int nr_el, liber[NMAX];

int este_plina()
{return nr_el == NMAX;}

int este_vida(adresa prim)
{return prim==NULL;}

void init()
{nr_el=0;
 for (adresa p=1;p<=NMAX;p++) liber[p]=1;}

adresa aloc_mem()
{adresa p;
```

```

for (p=1; !liber[p]; p++);
liber[p]=0; nr_el++;
return p;}

void adaug_prim(adresa &prim, adresa &ultim, int n)
{prim=ultim=alloc_mem();
 lista[prim].nr=n; lista[prim].urm=NULL;}

void adaug_inainte_prim(adresa &prim, int n)
{adresa p=alloc_mem();
 lista[p].nr=n; lista[p].urm=prim; prim=p;}

void adaug_dupa_ultim(adresa &ultim, int n)
{adresa p=alloc_mem();
 lista[p].nr=n; lista[p].urm=NULL; lista[ultim].urm=p; ultim=p;}

void creare(adresa &prim)
{adresa ultim; unsigned n;
 cout<<"Scrieti numerele:"<<endl; cout<<"numar= "; cin>>n;
 if (n!=0)
 {adaug_prim(prim,ultim,n);
 cout<<"numar= "; cin>>n;
 while (n!=0 && !este_plina())
 {if (n%2==0) adaug_inainte_prim(prim,n);
 else adaug_dupa_ultim(ultim,n);
 cout<<"numar= "; cin>>n;}}

void afisare(adresa p)
{while (p!=NULL) {cout<<lista[p].nr<<" "; p=lista[p].urm;}

void main()
{adresa prim; init(); creare(prim);
 if (!este_vida(prim)) {cout<<"Numerele sunt: ";
 afisare(prim);}
 else cout<<"Nu exista numere";}

```

### Temă



Refaceți programul folosind prototipurile funcțiilor. Explicați modul în care a fost descompusă problema în subprobleme și precizați specificațiile fiecărui modul.

**Enunțul problemei 3:** Se citește de la tastatură mai multe numere, până se citește valoarea 0. Numerele se scriu într-o listă, ordonate crescător. Să se afișeze numerele din nodurile listei.

Numerele care trebuie memorate în nodurile listei se citesc de la tastatură, până se citește valoarea 0 (are semnificația că nu mai există informație de adăugat). Nodurile acestei liste fiind ordonate, inserarea unui nou număr citit de la tastatură trebuie să se facă într-o poziție care să păstreze ordinea elementelor. Acest algoritm se numește **algoritm de sortare prin inserare**. Se pornește de la lista care conține primul nod cu informație. Pașii executați în acest algoritm sunt:

**PAS1.** Se adaugă primul nod cu informație la listă.

**PAS2.** Cât timp mai există informație de adăugat **execută**

**PAS3.** Se caută nodul înaintea căruia trebuie adăugat noul nod.

**PAS4.** Dacă s-a ajuns la sfârșitul listei, **atunci** se adaugă noul nod ca ultim nod din listă; **altfel**, se adaugă noul nod în fața nodului găsit. Se revine la Pasul 2.

#### Implementarea algoritmului.

```
#include <iostream.h>
const unsigned NMAX=100;
typedef unsigned adresa;
struct nod {int info;
 adresa urm;};
nod lista[NMAX+1];
int nr_el, liber[NMAX];

int este_plina()
{return nr_el==NMAX;}

int este_vida(adresa prim)
{return prim==NULL;}

void init()
{nr_el=0;
 for (adresa p=1;p<=NMAX; p++) liber[p]=1;}

adresa aloc_mem()
{adresa p; for (p=1; !liber[p]; p++);
 liber[p]=0; nr_el++;
 return p;}

void adaug_prim(adresa &prim, int n)
{prim=aloc_mem();
 lista[prim].info=n; lista[prim].urm=NULL;}

void adauga_ultim(adresa q, int n)
{adresa p=aloc_mem();
 lista[p].info=n; lista[q].urm=p; lista[p].urm=NULL;}

adresa cauta(adresa prim, int n)
{adresa p=prim;
 while(lista[p].info<n && lista[p].urm!=NULL) p=lista[p].urm;
 return p;}

void adauga_in_fata(adresa q, int n)
{adresa p=aloc_mem();
 lista[p].urm=lista[q].urm; lista[q].urm=p;
 lista[p].info=lista[q].info; lista[q].info=n;}

void creare(adresa &prim)
{adresa q; int n; init(); cout<<"numar "; cin>>n;
 adaug_prim(prim,n); //Se adaugă primul nod
 cout<<"numar "; cin>>n;
 while(n!=0 && !este_plina())
 {q=cauta(prim,n);
 //Se caută nodul q înaintea căruia trebuie adăugat nodul p
 if (lista[q].info<n) // Dacă s-a ajuns la sfârșitul listei
 adauga_ultim(q,n); //atunci nodul p se adaugă ca ultim nod
```

```

else adauga_in_fata(q,n); //altfel nodul p se adaugă în fața nodului q
cout<<"numar "; cin>>n;}}

void afiseaza(adresa prim)
{for (adresa p=prim; p!=NULL; p=lista[p].urm)
 cout<<lista[p].info<<" ";}

void main()
{adresa prim; creare(prim);
 afiseaza(prim);
 getch();
 clrscr();}
if (!este_vida) afiseaza(prim); //se parcurge lista pentru afișare
else cout<<"Nu exista elemente";}

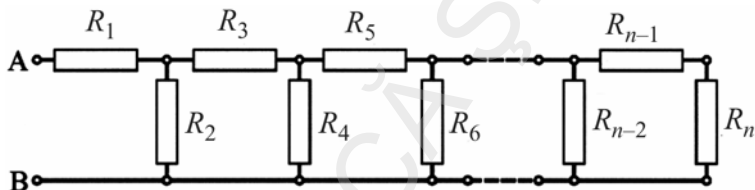
```

### Temă



Refaceți programul de la problema 2 astfel încât numerele din listă să fie afișate astfel: mai întâi numerele pare, ordonate crescător, și apoi numerele impare, ordonate descrescător.

**Enunțul problemei 2 – Calcularea rezistenței echivalente.** Să se calculeze rezistența echivalentă între punctele A și B pentru circuitul electric din figură.



Pentru calcularea rezistenței echivalente se pornește de la ultimele rezistențe –  $R_n$  și  $R_{n-1}$  – care sunt legate în serie. Se calculează rezistența lor echivalentă  $R_{e1}$ , care va fi legată în paralel cu rezistența  $R_{n-2}$ . Prin calcularea rezistenței echivalente a celor două rezistențe legate în paralel  $R_{e1}$  și  $R_{n-2}$ , se va obține o nouă rezistență echivalentă  $R_{e2}$  care este legată în serie cu rezistența  $R_{n-3}$ . Calcularea rezistenței echivalente a circuitului electric este un proces repetitiv în care alternează calcularea unei rezistențe echivalente a două rezistențe legate în serie cu calcularea unei rezistențe echivalente a două rezistențe legate în paralel. Pentru a ști care dintre variantele de calcul se alege, se folosește variabila  $s$ , care are valoarea 1 dacă rezistențele sunt legate în serie și valoarea 0 dacă sunt legate în paralel. Valorile pentru rezistențe se citesc dintr-un fișier text în care sunt memorate pe același rând, separate prin spațiu. Se creează o listă simplu înlănțuită în care ordinea de acces este inversă față de cea în care sunt citite numerele din fișier. Pentru calcularea rezistenței echivalente, lista se parcurge de la primul nod până la ultimul nod.

```

#include<fstream.h>
const unsigned NMAX=100;
typedef unsigned adresa;
struct nod {float info;
 adresa urm;};
nod lista[NMAX+1];
int nr_el, liber[NMAX];
fstream f("lista r.txt", ios::in);

int este_plina()
{return nr_el == NMAX;}

```

```

void init(adresa &prim)
{prim=NULL; nr_el=0;
 for (adresa p=1;p<=NMAX; p++) liber[p]=1;}

adresa aloc_mem()
{adresa p;
 for (p=1; !liber[p]; p++);
 liber[p]=0; nr_el++;
 return p;}

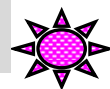
void adaug_inainte_prim(adresa &prim, float x)
{adresa p; p=aloc_mem();
 lista[p].info=x; lista[p].urm=prim; prim=p;}

void creare(adresa &prim)
{float x; init(prim);
 while (f>>x && !este_plina()) adaug_inainte_prim(prim,x);}

float R(adresa prim)
{int s=1; float r=lista[prim].info; adresa p=lista[prim].urm;
 while (p!=NULL)
 if (s) {r+=lista[p].info; s=0; p=lista[p].urm;}
 else {r=(r*lista[p].info)/(r+lista[p].info);
 s=1; p=lista[p].urm;}
 return r;}

void main()
{adresa prim; creare(prim); f.close();
 cout<<"Rezistenta echivalenta= "<<R(prim);}

```



### Temă



Scriveți câte un program care să rezolve cerințele fiecărei probleme. Pentru următoarele 8 probleme – în nodurile listelor se memorează numere întregi. Șirul de numere se citește dintr-un fișier text în care sunt scrise pe același rând, separate prin spațiu. După executarea unei operații de prelucrare a **listei simplu înlanțuite**, se vor afișa numerele din nodurile listei pentru a se verifica dacă operația de prelucrare s-a executat corect. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui.

1. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se mai citește un număr  $n$  de la tastatură. Se afișează elementul cu numărul de ordine  $n$  din listă. Dacă nu există, se afișează un mesaj de informare.
2. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se inserează, înaintea fiecărui număr, factorii săi primi.
3. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se inserează, între fiecare pereche de noduri, cel mai mare divizor comun al celor două numere.
4. Se creează două liste în care ordinea de acces este cea în care sunt citite numerele din fișiere. Se creează a treia listă prin concatenarea listei care are cele mai puține elemente, la lista care are cele mai multe elemente. Dacă listele au același număr de elemente, se va adăuga lista a doua la prima listă.
5. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se afișează numerele care au ultimele trei cifre identice, se elimină din listă numerele care au ultimele trei cifre consecutive și se inserează valoarea 10 înaintea numerelor care au suma ultimelor trei cifre egală cu 10.

6. Se creează o listă în care ordinea de acces este cea în care sunt citite numerele din fișier. Se afișează numerele care au mai mult de doi divizori primi, se inserează divizorii proprii în fața numerelor care au mai mult de trei divizori proprii și se elimină din listă numerele care au cel puțin două cifre identice.
7. Se creează o listă ordonată cu numerele din fișier. Să se inverseze ordinea de acces a nodurilor.
8. Se creează o listă ordonată cu numerele din fișier și se divizează apoi lista în două liste: una cu numere pare și una cu numere impare.
9. **Analiza lexicală a unui text.** Să se afișeze, în ordine alfabetică, cuvintele dintr-un text și frecvența de apariție a lor în text.

### 2.5.4. Algoritmi pentru prelucrarea stivelor

**Stiva** este o listă liniară în care operațiile de introducere și extragere se efectuează pe la o singură extremitate.

Această structură logică de date poate simula orice stivă de obiecte reale (un teanc de cărți, o stivă de farfurii etc.).

Cele două extremități ale stivei se numesc **bază** și **vârf**. Accesul la nodurile stivei (adăugarea, eliminarea sau consultarea unui nod) este permis numai prin extremitatea **vârf**. La o operație de adăugare a unui nod, vârful stivei urcă, iar la o operație de eliminare a unui nod vârful stivei coboară. Stiva se mai numește și structură **LIFO (Last In First Out)**, deoarece ultimul nod inserat este primul nod extras (se extrage cea mai nouă informație adăugată).

Implementarea stivei se poate face:

- Cu **alocare înlănțuită**. Implementarea stivei va fi la fel ca a unei liste liniare, cu deosebirea că, pentru adăugarea unui nod, se poate folosi numai **algoritmul de adăugare în fața primului nod**, iar pentru extragerea unui nod se poate folosi numai **algoritmul pentru eliminarea primului nod**.
- Cu **alocare secvențială**. Deoarece în stivă nu se pot executa operații în interiorul ei (adăugare sau eliminare de noduri), în prelucrarea stivei nu se vor mai folosi algoritmi de deplasare a elementelor din vector. Din această cauză, implementarea cu alocare secvențială poate fi folosită cu succes în cazul stivelor, furnizând un mecanism mult mai simplu – pentru adăugarea și eliminarea nodurilor.

Cea mai simplă implementare a stivelor este cea secvențială – cu ajutorul vectorului. Elementele vectorului pot fi date elementare sau structuri de date. Astfel, pentru implementarea stivei puteți folosi următoarele date și structuri de date:

```
const unsigned NMAX=100;
typedef <tip data> nod;
nod stiva[NMAX+1],val;
unsigned vf,baza;
```

- Variabilele **vf** și **baza** reprezintă indicii poziției din vector în care sunt memorate cele două extremități ale stivei: vârful și respectiv baza.
- Variabila **val** este de același tip ca și informația din nodul stivei (este folosită ca variabilă de manevră pentru informație).
- Tipul de dată **nod** poate fi un tip de bază al limbajului, în cazul în care nodurile stivei conțin date elementare sau un tip de structură de date (vector, șir de caractere sau înregistrare), în cazul în care nodurile stivei conțin structuri de date.

**Prelucrarea** unei stive se face **de la vârf spre bază**. Se prelucrează întotdeauna **nodul din vârful stivei**. **Accesul la informația** din acest nod se face cu **stiva[vf]**.

Există următoarele cazuri speciale de stive:

- **Stiva vidă** este stiva care nu conține nici un nod (indicele nodului **vf** are valoarea **NULL – 0**); întregul spațiu de memorie rezervat stivei este liber. În stiva vidă nu se mai pot executa operații de extragere de noduri. Stiva poate să ajungă să fie vidă în două cazuri:
  - a. la **inițializarea stivei**, când indicele bazei și al vârfului au valoarea **NULL – 0**;
  - b. după **extragerea ultimului nod**, când vârful devine mai mic decât baza și are valoarea 0.

Pentru testarea unei stive dacă este vidă, se poate implementa funcția operand **este\_vida()** – care va furniza valoarea 1 („adevărat”), dacă stiva este vidă, și valoarea 0 („fals”) dacă stiva nu este vidă.

```
int este_vida(unsigned vf)
{return vf==NULL;}
```

- **Stiva plină** este stiva pentru care spațiul de memorie rezervat este ocupat în întregime (indicele nodului **vârf** are valoarea lungimii fizice a vectorului). În stiva plină nu se mai pot executa operații de introducere de noduri. Pentru testarea unei stive dacă este plină, se poate implementa funcția operand **este\_plina()** – care va furniza valoarea 1 („adevărat”), dacă stiva este plină, și valoarea 0 („fals”) dacă stiva nu este plină.

```
int este_plina(unsigned vf)
{return vf==NMAX;}
```

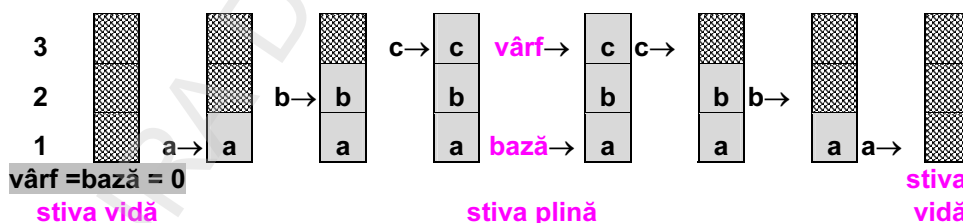
Pentru prelucrări cu ajutorul stivei puteți folosi următorii algoritmi:

- **inițializarea stivei**;
- **adăugarea unui nod la stivă**;
- **extragerea unui nod din stivă**;
- **consultarea nodului din vârful stivei**.

De exemplu, pentru stiva implementată astfel:

```
const unsigned NMAX=3;
typedef char nod;
nod stiva[NMAX+1],c;
unsigned vf,baza;
```

operațiile de adăugare și de extragere a caracterelor din stivă se execută astfel:



Se observă că:

- Dacă se introduc, în ordine, caracterele 'a', 'b' și 'c', se extrag în ordine caracterele 'c', 'b' și 'a'.
- Adăugarea unui nod în stivă incrementează cu 1 vârful stivei (vf++).
- Eliminarea unui nod din stivă decrementează cu 1 vârful stivei (vf--).



### 2.5.4.1. Inițializarea stivei

Prin acest algoritm **se creează stiva vidă**. În acest caz, nodul **vârf** nu există, și indicele **vf** va avea valoarea **NULL** – 0 (indicele 0 din vector; la fel ca și la listă, indicii pentru elementele stivei vor începe cu valoarea 1).

**Implementarea algoritmului.** Se folosește funcția procedurală **init()** al cărei parametru **vf** se transmite prin referință, deoarece este parametru de ieșire.

```
void init(unsigned &vf)
{vf = NULL;}
```

### 2.5.4.2. Adăugarea unui nod la stivă

Adăugarea unui nod la stivă se poate face numai dacă stiva nu este plină. Nodul adăugat este succesorul nodului curent și va fi nodul din vârful stivei. Pașii algoritmului sunt:

**PAS1.** Se alocă spațiu de memorie pentru nodul care se adaugă, prin incrementarea vârfului stivei.

**PAS2.** Se atribuie, nodului din vârful stivei, noua informație.

**Implementarea algoritmului.** Se folosește funcția procedurală **adauga()** ai cărei parametri se transmit astfel: **vf** prin referință, deoarece este parametru de intrare-ieșire, și **val** prin valoare, deoarece este parametru de intrare.

```
void adauga(unsigned &vf, nod val)
{if (!este_plină(vf)) {vf++; stiva[vf]=val;}}
```

### 2.5.4.3. Extragerea unui nod din stivă

Nu se poate elimina un nod decât în cazul în care stiva nu este vidă. Nodul se elimină prin coborârea vârfului în stivă (se eliberează spațiul care a fost ocupat de nod). Nodul din vârful stivei va fi predecesorul nodului eliminat.

**Implementarea algoritmului.** Se folosește funcția procedurală **extrage()** al cărei parametru **vf** se transmite prin referință, deoarece este parametru de intrare-ieșire.

```
void extrage(unsigned &vf)
{if (!este_vidă(vf)) vf--;}
```



Într-o stivă vidă, se adaugă următoarele numere naturale: 8, 12, 25, 2, 6, 5, 30, 20 și se extrag numerele până când primul număr prim ajunge în vârful stivei. Reprezentați grafic modul în care se execută operațiile.

### 2.5.4.4. Prelucrarea stivei

Prin acest algoritm se consultă informația din fiecare nod al stivei. Deoarece nu poate fi **consultată decât informația din vârful stivei**, pentru a ajunge la un nod trebuie să se extragă toate nodurile până la el.

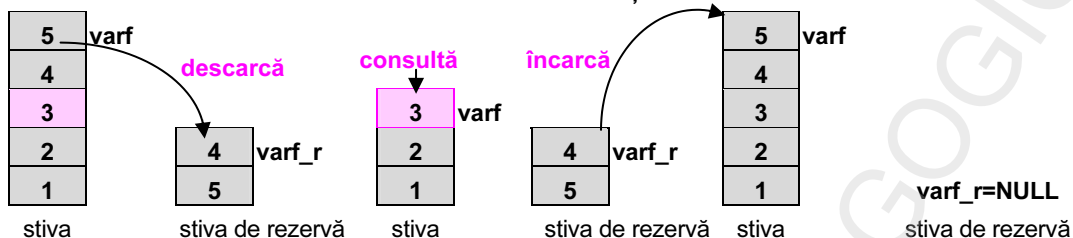
**Implementarea algoritmului.** Se folosește funcția procedurală **prelucrare()** al cărei parametru **vf** se transmite prin referință, deoarece este parametru de intrare-ieșire.

```
void prelucrare(unsigned &vf)
{while (!este_vidă(vf)) {//se prelucrează stiva[vf]
 extrage(vf);}}
```

#### Atenție

Pentru a ajunge la informația care trebuie prelucrată din stivă, trebuie extrase toate nodurile până la nodul care conține acea informație. Dacă nu vrem să se piardă informația din nodurile extrase, ele vor fi descărcate într-o altă stivă (de rezervă), iar după prelucrarea informației, nodurile vor fi încărcate din nou în stivă.

## Consultarea nodului care conține numărul 3



**Observație.** În construirea unei soluții prin **metoda backtracking**, s-a folosit de fapt o stivă:

- pentru adăugarea unui nou element al soluției, se urca în stivă;
- pentru revenirea la elementul anterior al soluției, se cobora în stivă;
- pentru găsirea unui element al soluției se prelucreză valoarea din vârful stivei.

### Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi algoritmii pentru prelucrarea stivelor pentru a rezolva probleme.

**Enunțul problemei 1:** Se citesc într-o stivă cifrele unui număr care are o valoare foarte mare. Citirea cifrelor se va face până când numărul citit nu este o cifră. Să se afișeze inversul acestui număr.

Se vor executa următorii pași:

**PAS1.** Se creează stiva vidă.

**PAS2.** Cât timp numărul citit în variabila *cif* este o cifră și stiva nu este plină **execută:** se adaugă la stivă un nod cu cifra citită.

**PAS3.** Se extrag nodurile din stivă, de la vârf spre bază, și se afișează informația din fiecare nod.

```
#include <iostream.h>
const unsigned NMAX=100;
typedef unsigned nod;
nod stiva[NMAX+1];

int este_vida(unsigned vf) {return vf==NULL;}
int este_plina(unsigned vf) {return vf==NMAX;}

void init(unsigned &vf) {vf = NULL;}

void adauga(unsigned &vf, nod val)
{if (!este_plina(vf)) {vf++; stiva[vf]=val;}}

void extrage(unsigned &vf)
{if (!este_vida(vf)) vf--;}

void prelucrare(unsigned &vf)
{while (!este_vida(vf)) {cout<<stiva[vf]; extrage(vf);}}

void main()
{nod cif; unsigned vf; init(vf);
 cout<<"Scrieti cifrele numarului:"<<endl;
```

```
cout<<"cifra= "; cin>> cif;
while (cif<=9 && !este_plina(vf))
 {adauga(vf, cif);
 cout<<"cifra= "; cin>> cif;}
cout<<"Numarul invers este: "; prelucrare(vf);}
```

### Temă

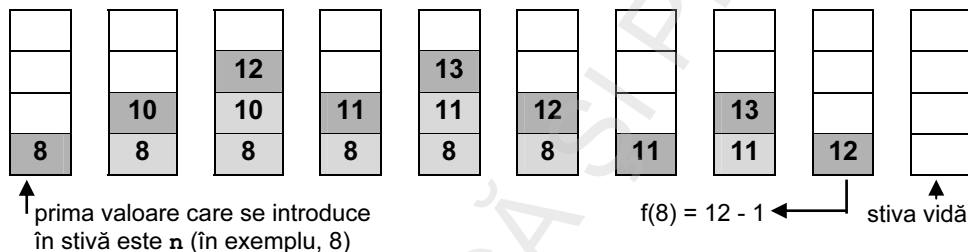


Scrieți un program prin care se citesc  $n$  cuvinte de la tastatură ( $n$  este o valoare care se citește de la tastatură) și se afișează în ordinea inversă citirii. Nu se vor folosi vectori de cuvinte.

**Enunțul problemei 2:** Să se calculeze valorile funcției Manna Pnuell, printr-un algoritm implementat iterativ.

Pentru a rezolva această problemă, va trebui să folosim o structură de date de tip stivă, în care se vor calcula valorile funcției pentru o valoare precizată,  $n$ , astfel:

- pentru cazul general al soluției (autoapelul funcției) – se urcă în stivă;
- pentru cazul de bază al soluției – se coboară în stivă.



Pașii algoritmului sunt:

**PAS1.** Se inițializează stiva, astfel: pe primul nivel din stivă ( $vf=1$ ) se introduce  $n$ , valoarea pentru care se calculează funcția – corespunde primului apel al funcției.

**PAS2.** Se analizează valoarea din vârful stivei ( $stiva[vf]$ ):

- Dacă** valoarea este mai mică decât 12, **atunci** are loc un autoapel al funcției – se urcă în stivă ( $vf++$ ) și se introduce în stivă noua valoare a argumentului funcției – este egală cu valoarea anterioară a argumentului incrementată cu 2:  $stiva[vf]=stiva[vf-1]+2$ .
- Dacă** valoarea este mai mare sau egală cu 12, **atunci** funcția se poate calcula – se coboară în stivă ( $vf--$ ) și se introduce pe acest nivel noua valoare a argumentului funcției, care este egală cu valoarea anterioară a argumentului decrementată cu 1:  $stiva[vf] = stiva[vf+1]-1$ .

**PAS3.** Dacă stiva nu este vidă, **atunci** se revine la **Pas2**; **altfel**, se trece la **Pas4**.

**PAS4.** Se calculează valoarea funcției ca fiind ultima valoare a argumentului din stivă, decrementată cu 1:  $stiva[1]-1$ .

```
#include <iostream.h>
const unsigned NMAX=100;
typedef unsigned nod;
nod stiva[NMAX+1];

int este_vida(unsigned vf) {return vf==NULL;}

int este_plina(unsigned vf) {return vf==NMAX;}

void init(unsigned &vf,nod nr) {vf=1; stiva[vf]=nr;}
```



prin nodul **ultim**, iar extragerea unui nod este permisă numai prin extremitatea **prim**. Coada se mai numește și structură **FIFO (First In First Out)**, deoarece primul nod inserat este primul nod extras (se extrage cea mai veche informație adăugată).

Implementarea cozii se poate face:

- Cu **alocare înlănțuită**. Implementarea cozii va fi la fel ca a unei liste liniare, cu deosebirea că, pentru adăugarea unui nod, se poate folosi numai **algoritmul de adăugare după ultimul nod**, iar pentru extragerea unui nod se poate folosi numai **algoritmul pentru eliminarea primului nod**.
- Cu **alocare secvențială**. Ca și în cazul stivei, deoarece nu se pot executa operații în interiorul cozii, această implementare oferă un mecanism mult mai simplu pentru prelucrarea cozilor.

Cea mai simplă implementare a cozii este cea secvențială, cu ajutorul vectorului, pentru care puteți folosi următoarele date și structuri de date:

```
const unsigned NMAX=100;
typedef <tip data> nod;
nod coada[NMAX+1],val;
unsigned ultim,prim;
```

- Variabilele **prim** și **ultim** reprezintă indicii poziției din vector în care sunt memorate cele două extremități ale structurii: capul și respectiv sfârșitul cozii.
- Variabila **val** este de același tip ca și informația din nodul cozii (este folosită ca variabilă de manevră pentru informație).

**Prelucrarea unei cozi se face de la nodul prim spre nodul ultim.** Se prelucreează întotdeauna **nodul prim**. **Accesul la informația** din acest nod se face cu **coada[prim]**.

Există următoarele cazuri speciale de cozi:

- **Coada vidă** este coada care nu conține nici un nod (indicele nodului **ultim** are valoarea **NULL – 0**); întregul spațiu de memorie rezervat cozii este liber. În coada vidă nu se mai pot executa operații de extragere de noduri. Coada poate să ajungă să fie vidă în două cazuri:
  - a. la **inițializare**, când indicii nodurilor **prim** și **ultim** au valoarea **NULL**;
  - b. după **extragerea ultimului nod**, când indicele nodului **ultim** devine mai mic decât indicele nodului **prim**.

Pentru testarea unei cozi dacă este vidă, se poate implementa funcția operand **este\_vida()** – care va furniza valoarea 1 („adevărat”), dacă este vidă, și valoarea 0 („fals”) dacă nu este vidă.

```
int este_vida(unsigned prim, unsigned ultim)
{return (ultim==NULL) || (ultim<prim);}
```

- **Coada plină** este coada pentru care spațiul de memorie rezervat este ocupat în întregime (nodul **ultim** are valoarea lungimii fizice a vectorului). În coada plină nu se mai pot executa operații de adăugare de noduri. Pentru testarea unei cozi dacă este plină se poate implementa funcția operand **este\_plina()** – care va furniza valoarea 1 („adevărat”), dacă este plină, și valoarea 0 („fals”) dacă nu este plină.

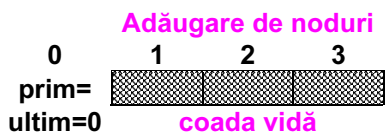
```
int este_plina(unsigned ultim)
{return ultim==NMAX;}
```

De exemplu, pentru coada implementată astfel:

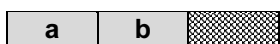
```
const unsigned NMAX=3;
typedef char nod;
```

```
nod coada[NMAX+1], c;
unsigned prim, ultim;
```

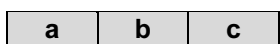
operațiile de adăugare și de extragere a caracterelor din coadă se execută astfel:



prim=ultim=1

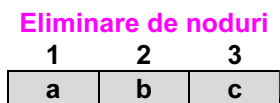


prim      ultim

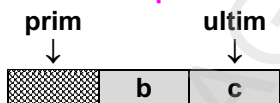


prim      ultim

coada plină



coada plină



prim      ultim



ultim=

prim

ultim=

prim



ultim

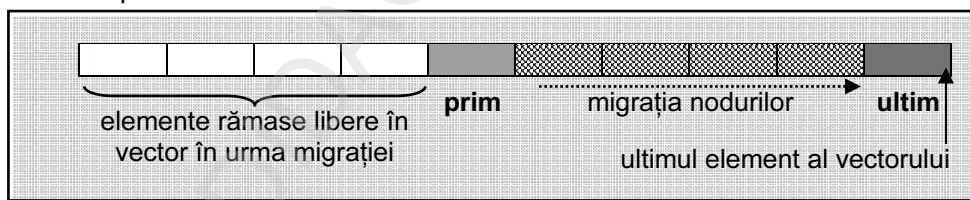
prim

coada vidă

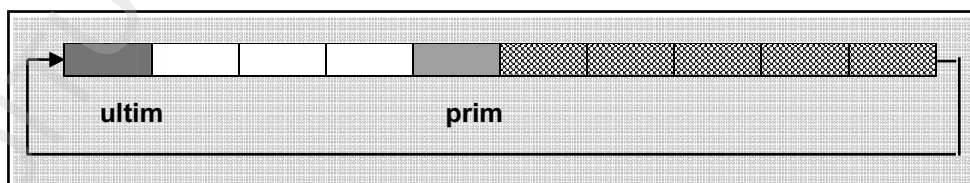
Se observă că:

- Dacă se introduc, în ordine, caracterele 'a', 'b' și 'c', se extrag în ordine caracterele 'a', 'b' și 'c'.
- Adăugarea unui nod în coadă incrementează cu 1 nodul **ultim** (ultim++).
- Eliminarea unui nod din coadă incrementează cu 1 nodul **prim** (prim++).

Operațiile de adăugare și extragere a nodurilor din coadă vor face să migreze atât nodul **ultim** cât și nodul **prim**, către ultimul element din vector, rămânând la începutul vectorului locuri libere. În acest mod, coada va ajunge foarte repede la capătul spațiului alocat și nu va mai putea fi folosită.



Soluția pentru acest neajuns este de a implementa coada ca pe o listă circulară: după ce ultimul nod al cozii a ocupat ultimul element al vectorului, la următoarea operație de adăugare a unui nod, acesta va fi inserat în primul element al vectorului. În același mod, la eliminarea unui nod din coadă, dacă a fost eliminat nodul din ultimul element al vectorului, la următoarea operație de eliminare se va elimina nodul din primul element al vectorului.



Prin acest mod de implementare, nu mai pot fi folosite expresiile anterior definite pentru a verifica starea cozii: dacă este vidă sau dacă este plină. Vom folosi o variabilă de memorie suplimentară **k** în care vom număra nodurile cozii. La inițializarea cozii, **k** are valoarea 0, la adăugarea unui nod este incrementat cu 1, iar la eliminarea unui nod este decrementat cu 1.

Funcția **este\_vida()** va fi definită astfel:

```
int este_vida(unsigned k) {return k==0;}
```

iar funcția **este\_plina()** va fi definită astfel:

```
int este_plina(unsigned k) {return k==NMAX;}
```

Pentru prelucrări cu ajutorul cozii puteți folosi următorii algoritmi:

- inițializarea cozii;
- adăugarea unui nod la coadă;
- extragerea unui nod din coadă;
- consultarea nodului de la capul cozii.

### 2.5.5.1. Inițializarea cozii

Prin acest algoritm **se creează coada care conține un nod**. În acest caz, nodurile **prim** și **ultim** vor avea același indice (1), iar numărul de noduri din coadă, **k**, va fi egal cu 1.

**Implementarea algoritmului.** Se folosește funcția procedurală **init()** ai cărei parametri **prim**, **ultim** și **k** se transmit prin referință, deoarece sunt parametri de ieșire.

```
void init(unsigned &prim, unsigned &ultim, unsigned &k, nod val)
{prim=ultim=1; k=1; coada[ultim]=val;}
```

### 2.5.5.2. Adăugarea unui nod la coadă

Adăugarea unui nod la coadă se poate face numai dacă ea nu este plină. În acest caz pașii executați pentru acest algoritm sunt:

- PAS1.** Se alocă spațiu pentru noul nod: **dacă** ultimul element din coadă a ajuns la capătul vectorului, **atunci** se inițializează cu 1; **altfel**, se incrementează cu 1.
- PAS2.** Se atribuie, nodului adăugat, informație.
- PAS3.** Se incrementează cu 1 numărul de elemente din coadă.

**Implementarea algoritmului.** Se folosește funcția procedurală **adauga()** ai cărei parametri **ultim** și **k** se transmit prin referință, deoarece sunt parametri de intrare-ieșire.

```
void adauga(unsigned &ultim, unsigned &k, nod val)
{if (!este_plina(k))
{if (ultim==NMAX) ultim=1; else ultim++;
coada[ultim]=val; k++;}}
```

### 2.5.5.3. Extragerea unui nod din coadă

Nu se poate elimina un nod decât în cazul în care coada nu este vidă. Pașii algoritmului sunt:

- PAS1.** Se eliberează spațiul ocupat de nod: **dacă** primul element din coadă a ajuns la capătul vectorului, **atunci** se inițializează cu 1; **altfel**, se incrementează cu 1.
- PAS2.** Se decrementează cu 1 numărul de elemente din coadă.

**Implementarea algoritmului.** Se folosește funcția procedurală **extrage()** ai cărei parametri **prim** și **k** se transmit prin referință, deoarece sunt parametri de intrare-ieșire.

```
void extrage(unsigned &prim, unsigned &k)
{if (!este_vida(k))
```



```
{if (prim==NMAX) prim=1; else prim++;
 k--;}}
```

### Temă



Într-o coadă vidă se adaugă următoarele numere naturale: 1, 2, 25, 27, 3, 8, 30 și se extrag numerele până la primul număr care este un cub perfect. Reprezentați grafic modul în care se execută operațiile.

### 2.5.5.4. Prelucrarea cozii

Prin acest algoritm se consultă informația din fiecare nod al cozii. Deoarece nu poate fi **consultată decât informația din capul cozii**, pentru a ajunge la un nod trebuie să se extragă toate nodurile până la el.

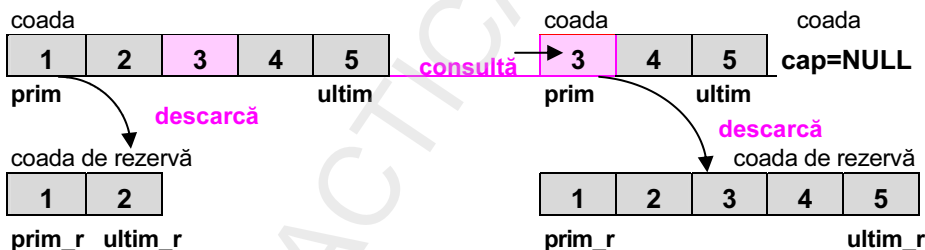
**Implementarea algoritmului.** Se folosește funcția procedurală **prelucrare()** ai cărei parametri **prim** și **k** se transmit prin referință, deoarece sunt parametri de intrare-ieșire.

```
void prelucrare(unsigned &prim, unsigned &k)
{while (!este_vida(k)) { //se prelucrează coada[prim]
 extrage(prim, k); }}
```

### Atenție

În prelucrarea cozilor, pentru a putea ajunge la informația care trebuie prelucrată, trebuie extrase toate nodurile până la nodul care conține acea informație, deoarece nu poate fi prelucrată decât informația din capul cozii. Dacă nu vrem să pierdem informația din nodurile extrase, ele vor fi descărcate într-o altă coadă (de rezervă), iar după prelucrarea informației, nodurile vor fi descărcate în continuare în coada de rezervă – până când coada inițială devine vidă.

#### Consultarea nodului care conține numărul 3



### Studiu de caz

**Scop:** exemplificarea modului în care puteți folosi algoritmi pentru prelucrarea cozilor, pentru a rezolva probleme.

**Enunțul problemei:** Se adaugă și se extrag dintr-o coadă mai multe caractere.

Executați următorul program. Reprezentați grafic modul în care se execută operațiile. Înlocuiți în program, în instrucțiunea (1), expresia  $i \leq 2$  cu  $i \leq 3$ . Executați programul după ce ați făcut modificarea. Ce constatați? În ce stare este coada? Înlocuiți în program, în instrucțiunea (2), expresia  $i \leq 4$  cu  $i \leq 5$ . Executați programul după ce ați făcut modificarea. Ce constatați? În ce stare este coada?

```
#include <iostream.h>
const unsigned NMAX=3;
typedef char nod;
nod coada[NMAX+1],n;
```

```

int este_vida(unsigned k) {return k==0;}
int este_plina(unsigned k) {return k==NMAX;}
void init(unsigned &prim, unsigned &ultim, unsigned &k, nod val)
{prim=ultim=1; k=1; coada[ultim]=val;}
void adauga(unsigned &ultim, unsigned &k, nod val)
{if (!este_plina(k))
 {if (ultim==NMAX) ultim=1; else ultim++;
 coada[ultim]=val; k++;}}
void extrage(unsigned &prim, unsigned &k)
{if (!este_vida(k))
 {if (prim==NMAX) prim=1; else prim++;
 k--;}}
void main()
{unsigned prim,ultim,k,i;
 init(prim,ultim,k,'a');
 for (i=1;i<=3;i++)
 if (!este_plina(k)) adauga(ultim,k,'a'+i);
 for (i=1;i<=2;i++)
 if (!este_vida(k)) {cout<<coada[prim]<<" "; extrage(prim,k);}
 for (i=1;i<=2;i++) // (1)
 if (!este_plina(k)) adauga(ultim,k,'A'+i);
 for (i=1;i<=4;i++) // (2)
 if (!este_vida(k)) {cout<<coada[prim]<<" "; extrage(prim,k);}
 cout<<endl<<ultim<<" " <<prim;}

```

### Temă



Scriveți câte un program care să rezolve cerințele fiecărei probleme. Se creează **cozi** în nodurile cărora se memorează numere



întregi. Șirul de numere se citește dintr-un fișier text în care sunt memorate pe același rând, separate prin spațiu. Se vor alege seturi de date de intrare astfel încât să se verifice algoritmul pe toate traseele lui.

1. Să se elimine din coadă numerele pare.
2. Să se concateneze două cozi, adăugând a doua coadă la sfârșitul primei cozi.
3. Să se verifice dacă numerele dintr-o coadă sunt ordonate (crescător sau descrescător).
4. Să se inverseze ordinea numerelor dintr-o coadă cu ajutorul unei stive.
5. Există două automate care eliberează bonuri de ordine pentru o coadă de așteptare. Pe fiecare bon de ordine este trecut numărul bonului și momentul la care a fost eliberat (exprimat în oră, minut și secundă). Bonurile au numere unice. Deservirea persoanelor se face în ordinea momentului la care s-au eliberat bonurile. Să se organizeze o coadă de așteptare pe baza bonurilor emise de cele două automate.

## Evaluare

Pentru exercițiile de mai jos, se vor folosi următoarele date:

```

struct nod
{int info; unsigned urm;};
nod lista[100]; unsigned p,q,r,i,k;
// p,q și r sunt adresele (indicii) unor noduri din listă

```

### Răspundeți:

1. O listă simplu înlănțuită conține, în ordine, următoarele noduri:  $2 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 18 \rightarrow 20$ . Ce se va afișa în urma execuției următoarei secvențe de program?

```
for (p=prim, k=1; k<5; k++) p=lista[p].urm;
cout<<lista[p].info;
```

2. O listă simplu înlănțuită conține, în ordine, următoarele noduri:  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$ . Ce se va afișa în urma execuției următoarei secvențe de program?

```
for (k=0, p=prim; lista[p].urm!=NULL; p=lista[p].urm)
 if (lista[p].info%2) k+=lista[p].info;
cout<<k;
```

3. Ce se va afișa în urma execuției următoarei secvențe de program, dacă lista simplu înlănțuită conține în ordine următoarele noduri:  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ ? Dar dacă nodurile ei sunt, în ordine:  $1 \rightarrow 0 \rightarrow 3 \rightarrow 0 \rightarrow 5$ ?

```
for (p=prim; lista[p].info!=0; p=lista[p].urm);
cout<<lista[p].info;
```

4. Ce se va afișa în urma execuției următoarei secvențe de program, dacă lista simplu înlănțuită conține, în ordine, următoarele noduri:  $1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9$ ? Dar dacă nodurile ei sunt, în ordine:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8$ ?

```
for (k=0, p=prim; lista[p].info!=0; p=lista[p].urm)
 if (lista[lista[p].urm].info - lista[p].info == 1) k++;
cout<<k;
```

### Adevărat sau Fals:

- Dacă **p** este primul nod al listei, pentru a afișa informația din al doilea nod se execută secvența de instrucțiuni: `p=lista[p].urm; cout<<lista[p].info;`
- Dacă **p** este primul nod al listei, pentru a afișa informația din al doilea nod se execută instrucțiunea: `cout<<lista[lista[p].urm].info;`
- Stiva este o listă restrictivă în care primul element introdus în listă este primul care poate fi extras.
- Coada este o listă restrictivă în care ultimul element introdus în listă este primul care poate fi extras.
- Într-o coadă, adăugarea unui nod se poate face numai în fața primului nod.
- Într-o stivă, adăugarea unui nod se poate face numai pe la baza stivei.
- Extragerea unui nod dintr-o stivă se poate face numai prin vârful ei.
- Dintr-o coadă se poate extrage numai ultimul nod.
- Într-o coadă aveți acces numai la primul și la ultimul nod.

### Alegeți:

1. Dacă **p** este primul nod al listei, care dintre următoarele expresii reprezintă nodul al treilea al listei?

- |                     |                            |
|---------------------|----------------------------|
| a. lista[p].urm     | b. lista[lista[p].urm].urm |
| c. lista[p].urm.urm | d. lista[p.urm].urm        |

2. Dacă **p** este primul nod al listei, iar **q** al doilea nod al listei, prin ce instrucțiune se leagă nodul **p** de nodul **q**?

- |                    |                    |
|--------------------|--------------------|
| a. p=q;            | b. p=lista[q].urm; |
| c. q=lista[p].urm; | d. lista[p].urm=q; |

3. Care dintre următoarele variante realizează corect legăturile, în cazul inserării unui nod nou într-o listă simplu înlănțuită, dacă nodul nou are adresa  $p$ , iar nodul după care se inserează are adresa  $q$ ?

a. `lista[p].urm=q; lista[q].urm=lista[p].urm;`  
 b. `lista[p].urm=lista[q].urm; lista[q].urm=p;`  
 c. `lista[q].urm=lista[p].urm;`  
 d. `lista[p].urm=lista[q].urm; lista[p].urm=q;`

4. Dacă  $p$  este primul nod al listei, ce instrucțiune trebuie executată pentru a tipări informația memorată în al treilea nod?

a. `cout<<lista[lista[p].info].info;`  
 b. `cout<<lista[lista[p].urm].info;`  
 c. `cout<<lista[lista[lista[p].urm].info].info;`  
 d. `cout<<lista[lista[lista[p].urm].urm].info;`

5. Dacă  $p$ ,  $q$  și  $r$  sunt trei noduri consecutive ale listei – pentru a interschimba nodul  $q$  cu nodul  $r$ , care dintre secvențele de instrucțiuni este corectă?

a. `lista[r].urm=q; lista[q].urm=lista[r].urm; lista[p].urm=r;`  
 b. `lista[p].urm=r; lista[r].urm=q; lista[q].urm=lista[r].urm;`  
 c. `lista[q].urm=lista[r].urm; lista[r].urm=q; lista[p].urm=r;`  
 d. `lista[r].urm=q; lista[p].urm=r; lista[q].urm=lista[r].urm;`

6. Dacă  $p$  este un nod al listei și  $k$  o variabilă intermediară, care dintre secvențele de instrucțiuni următoare realizează corect legăturile, astfel încât să se elimine din listă cele două noduri care urmează după nodul  $p$ ?

a. `lista[p].urm=lista[lista[lista[lista[p].urm].urm].urm].urm;`  
 b. `lista[p].urm=lista[lista[p].urm].urm;`  
 c. `lista[p].urm=lista[lista[lista[p].urm].urm].urm;`  
 d. `k=lista[p].urm; lista[p].urm=lista[lista[k].urm].urm;`

7. Care dintre următoarele secvențe de program calculează, în variabila  $k$ , suma elementelor din listă simplu înlănțuită:

a. `for(k=0,p=prim;p!=NULL;p=lista[p].urm) k+= lista[p].info;`  
 b. `for(k=0,p=prim;lista[p].urm!=NULL;p=lista[p].urm)`  
     `k+=lista[p].info;`  
 c. `k=0; p=prim;`  
     `while (p!=NULL) {k+= lista[p].info; p=lista[p].urm;}`  
 d. `p=prim; k=lista[p].info;`  
     `do {p=lista[p].urm; k+=lista[p].info;} while (lista[p].urm!=NULL);`

8. Dacă  $p$  este primul nod al listei, iar  $q$  ultimul nod al listei, care dintre următoarele instrucțiuni transformă listă simplu înlănțuită într-o listă circulară?

a. `q=p`                      b. `lista[q].urm=lista[p].urm;`  
 c. `lista[q].urm=p;`      d. `lista[p].urm=q;`

9. Dacă  $vf$  indică ultimul nod al stivei, care dintre următoarele expresii trebuie să fie adevărate, pentru ca stiva să fie vidă?

a. `vf==NULL`                      b. `stiva[vf].urm==0`  
 c. `stiva[vf].urm==NULL`      d. `vf==0`

10. Dacă  $vf$  indică ultimul nod al stivei, iar  $k$  este o variabilă intermediară, care dintre următoarele secvențe de instrucțiuni elimină nodul din vârful stivei?

a. `k=stiva[vf].urm;`                      b. `k=vf;`  
     `vf=stiva[k].urm;`                      `vf=stiva[vf].urm;`  
     `liber[k]=1;`                              `liber[vf]=1;`

c. `k=vf;`  
`vf=stiva[k].urm;`  
`liber[k]=1;`

a. `stiva[k]=stiva[vf];`  
`vf=stiva[k].urm;`  
`liber[k]=1;`

11. Dacă variabila **vf** memorează adresa nodului din vârful stivei, iar **baza** adresa nodului de la baza stivei, care dintre următoarele expresii trebuie să fie adevărate, pentru a avea o stivă vidă?

a. `baza==NULL`

b. `vf<baza`

c. `vf==baza`

d. `vf==NULL`

12. Dacă variabila **prim** memorează adresa primului nod din coadă și variabila **ultim** memorează adresa ultimului nod din coadă, care dintre următoarele expresii trebuie să fie adevărate, pentru a avea o coadă vidă?

a. `ultim==NULL`

b. `coada[prim].urm==ultim`

c. `prim==ultim`

d. `prim==NULL`

## Miniproiecte:

Pentru realizarea **miniproiectelor** se va lucra **în echipă**. Fiecare miniproiect va conține:

- descompunerea problemei în subprobleme și rezolvarea fiecărei subprobleme cu ajutorul unui subprogram.**
- explicarea algoritmilor folosiți pentru realizarea subprogramelor;
- citirea datelor de intrare dintr-un fișier text;
- un exemplu de problemă cu aplicare practică, în care, pentru implementarea datelor se vor folosi liste liniare – și rezolvarea acestor probleme (Exemplu. *La un concurs participă mai mulți candidați, identificați după nume și prenume. Fiecare candidat primește la înscriere un număr de identificare. Concursul constă în trei probe, notele putând lua valori de la 1 la 10. Rezultatul concursului se stabilește pe baza mediei aritmetice a notelor primite. Sunt admiși, în ordinea mediilor, primii **k** candidați – valoarea pentru **k** se citește de la tastatură. Programul trebuie să rezolve următoarele cerințe: înscrierea unui nou candidat la concurs, retragerea unui candidat din concurs, completarea notelor și calcularea mediei, pentru fiecare candidat, și afișarea candidaților admiși în ordinea descrescătoare a mediilor.*)

1. Să se afișeze cuvintele dintr-un text în ordinea inversă în care apar în text. Folosiți, pentru memorarea cuvintelor din text, o structură de date de tip:

a) listă liniară;

b) stivă;

c) vector de șiruri de caractere.

2. Să se realizeze conversia unui număr **nr** din baza 10 în baza **q**. Folosiți, pentru memorarea cifrelor numărului obținut în urma conversiei, o structură de date de tip:

a) listă liniară;

b) stivă;

c) coadă.

3. Să se concateneze două liste înălțuite:

a) ordonate;

b) neordonate.

4. Într-o listă, sunt memorate mai multe numere naturale. Să se divizeze această listă în două liste: una cu numerele pare și una cu numerele impare. Listele obținute vor fi:

a) ordonate;

b) neordonate.

5. Să se elimine cifrele pare, dintr-un număr cu maximum 20 de cifre, și să se afișeze numărul astfel obținut. Folosiți, pentru memorarea cifrelor numărului, o structură de date de tip:

a) listă liniară;

b) stivă;

c) vector.

6. Să se adune două numere, care au maximum 20 de cifre fiecare, și să se afișeze rezultatul obținut. Folosiți, pentru memorarea cifrelor numărului, o structură de date de tip:

a) listă liniară;

b) stivă;

c) vector.

7. Se ordonează crescător un șir de numere, cu ajutorul a două:

a) stive; b) cozi.

## 2.6. Graful

### 2.6.1. Definiția matematică a grafului

Se numește **graf (G)** o pereche ordonată de mulțimi  $(X, U)$ , unde  $X$  este o mulțime finită și nevidă, iar  $U$  o mulțime de **perechi** formate cu elemente **distincte** din mulțimea  $X$  (familie de submulțimi cu două elemente din mulțimea  $X$ ).

#### Terminologie:

→ Elementele mulțimii  $X$  se numesc **vârfuri** sau **noduri**. Mulțimea  $X$  se mai numește și **mulțimea vârfurilor** sau **mulțimea nodurilor** grafului  $G$ . Ea este de forma:

$$X = \{x_1, x_2, x_3, \dots, x_i, \dots, x_n\}$$

unde  $x_i$  reprezintă nodul  $i$  al grafului  $G$  care are  $n$  noduri.

→ **Ordinul grafului** reprezintă numărul de noduri ale grafului,  $n$ :

$$\text{ordinul grafului} = \text{card}(X) = n$$

→ Elementele mulțimii  $U$  sunt perechi de noduri, adică submulțimi cu două elemente din mulțimea  $X$ , și se notează cu  $u_k$ . Elementul  $u_k$  este definit de perechea de forma  $\{x_i, x_j\}$ , unde  $x_i, x_j \in X$  și  $x_i \neq x_j$  (elemente distincte din mulțimea  $X$ ). Elementul  $u_k$  leagă nodurile  $x_i$  și  $x_j$  și se notează astfel:  $[x_i, x_j]$ . Mulțimea  $U$  este de forma:

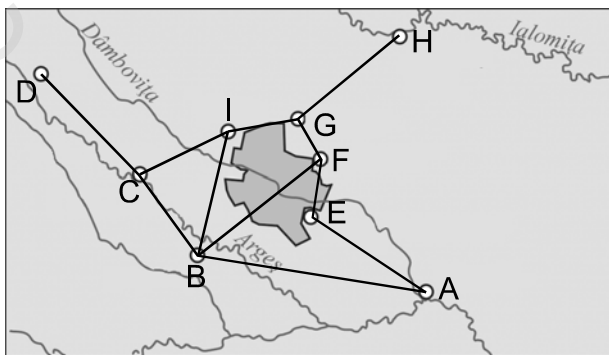
$$U = \{u_1, u_2, u_3, \dots, u_k, \dots, u_m\}$$

### Studiu de caz

**Scop:** exemplificarea modului în care identificați structura de date pe care o folosiți pentru a rezolva problema.

**Enunțul problemei.** O persoană dorește să viziteze șapte obiective turistice care se găsesc în șapte localități, pe care le notăm cu  $A, B, C, E, F, G$  și  $I$ . Între cele șase obiective turistice există mai multe căi de acces. Trebuie să se găsească un traseu, astfel încât turistul să plece din localitatea  $A$  și să revină în localitatea  $A$ , vizitând toate cele șapte obiective fără să treacă de două ori prin aceeași localitate.

Dacă există mai multe trasee, să se caute traseul cu drumul cel mai scurt.



Pentru rezolvarea problemei, trebuie stabilită structura de date care se va folosi. Localitățile reprezintă entități care pot fi legate sau nu prin căi de acces. Fiecare cale de acces este caracterizată de distanța dintre cele două localități. Pentru a reprezenta la nivel **conceptual** această structură de date, vom **reprezenta în plan** harta zonei turistice prin intermediul unor elemente geometrice astfel: localitățile se reprezintă prin cercuri în

interiorul cărora vom scrie identificatorul localității, iar căile de acces prin linii drepte care unesc anumite puncte. Acest model de reprezentare a datelor este un **graf**.



### Clasificarea grafurilor:

Criteriul de clasificare folosit este **proprietatea de simetrie** a mulțimii  $U$ .

**Mulțimea  $U$  are proprietatea de simetrie dacă și numai dacă pentru orice pereche de noduri  $(x_i, x_j)$ , dacă  $\{x_i, x_j\} \in U$ , atunci și  $\{x_j, x_i\} \in U$**

În funcție de proprietatea de simetrie, grafurile se clasifică în:

- **Grafuri neorientate**. Un graf  $G=(X,U)$  este un graf neorientat dacă mulțimea  $U$  are proprietatea de simetrie. Mulțimea  $U$  este formată din **perechi neordonate  $\{x_i, x_j\}$** .
- **Grafuri orientate**. Un graf  $G=(X,U)$  este un graf orientat dacă mulțimea  $U$  nu are proprietatea de simetrie. Mulțimea  $U$  este formată din **perechi ordonate  $\{x_i, x_j\}$** .

Pentru a identifica tipul de graf pe care îl veți folosi pentru a reprezenta datele, definiți relația dintre nodurile grafului și verificați dacă relația are proprietatea de simetrie, astfel:

- Dacă nodul  $x$  în relație cu nodul  $y$  implică și că nodul  $y$  este în relație cu nodul  $x$ , atunci graful este neorientat.
- Dacă nodul  $x$  în relație cu nodul  $y$  nu implică și că nodul  $y$  este în relație cu nodul  $x$ , atunci graful este orientat.

### Studiu de caz

**Scop:** identificarea tipului de graf pe care îl folosiți pentru a rezolva problema.

**Enunțul problemei 1.** *Pe harta unui județ există mai multe localități care sunt legate de șosele pe care se circulă în ambele sensuri. Să se identifice traseele prin care se poate ajunge de la localitatea A la localitatea B.*

Nodurile grafului sunt localitățile. Relația care se stabilește între nodurile grafului este: nodul  $x$  este în relație cu nodul  $y$ , dacă există o șosea care leagă direct localitatea asociată nodului  $x$  cu localitatea asociată nodului  $y$ . Relația are proprietatea de simetrie, deoarece șoseaua care leagă direct localitatea asociată nodului  $x$  cu localitatea asociată nodului  $y$  leagă direct și localitatea asociată nodului  $y$  cu localitatea asociată nodului  $x$ . Pentru reprezentarea căilor de comunicație dintre localități se va folosi un **graf neorientat**.

**Enunțul problemei 2.** *Pe harta unui cartier există mai multe intersecții care sunt legate de străzi. Pe unele străzi se poate circula în ambele sensuri, pe alte străzi numai într-un anumit sens. Să se identifice traseele prin care se poate ajunge de la intersecția A la intersecția B.*

Nodurile grafului sunt intersecțiile. Relația care se stabilește între nodurile grafului este: nodul  $x$  este în relație cu nodul  $y$ , dacă există trafic care leagă direct intersecția asociată nodului  $x$  cu intersecția asociată nodului  $y$  (se poate circula de la nodul  $x$  la nodul  $y$ ). Relația nu are proprietatea de simetrie deoarece, dacă există o stradă care leagă direct intersecția asociată nodului  $x$  cu intersecția asociată nodului  $y$  și pe această stradă există trafic de la nodul  $x$  la nodul  $y$ , nu este obligatoriu ca pe acea stradă să existe trafic și de la nodul  $y$  la nodul  $x$ . Pentru reprezentarea traficului auto dintre intersecții se va folosi un **graf orientat**.

**Enunțul problemei 3.** *La nivelul unui grup de persoane se face un studiu social. Între persoane se stabilesc relații de prietenie, dar și relații de simpatie. Să se descrie cu ajutorul grafului relațiile dintre persoane.*



Nodurile grafului sunt membrii grupului de persoane. Între persoane se pot stabili relațiile:

- Relația de prietenie este o relație definită astfel: persoana  $x$  este în relație cu persoana  $y$ , dacă este prietenă cu ea. Relația este simetrică deoarece, dacă persoana  $x$  este prietenă cu persoana  $y$ , atunci și persoana  $y$  este prietenă cu persoana  $x$  (relația de prietenie presupune reciprocitate). Pentru reprezentarea relațiilor de prietenie dintre membrii grupului se va folosi un **graf neorientat**.
- Relația de simpatie este o relație definită astfel: persoana  $x$  este în relație cu persoana  $y$ , dacă o simpatizează. Relația nu este simetrică deoarece, dacă persoana  $x$  simpatizează persoana  $y$ , atunci nu este obligatoriu ca persoana  $y$  să simpatizeze persoana  $x$  (relația de simpatie nu presupune reciprocitate). Pentru reprezentarea relațiilor de simpatie dintre membrii grupului se va folosi un **graf orientat**.

**Temă**



1. Prin ce tip de graf va fi reprezentat un grup de persoane între care s-au stabilit relații de vecinătate ?
2. Prin ce tip de graf va fi reprezentat un grup de persoane între care s-au stabilit relații de cunoștință ?



## 2.6.2. Graful neorientat

### 2.6.2.1. Terminologie

- Elementele mulțimii  $U$  (perechile de noduri) se numesc **muchii**. Mulțimea  $U$  se mai numește și **mulțimea muchiilor** grafului  $G$ . O muchie, fiind un element din mulțimea  $U$ , este determinată de o submulțime cu două elemente din mulțimea  $X$ : muchia  $k$  a grafului ( $u_k$ ), care unește nodurile  $x_i$  și  $x_j$ , este determinată de submulțimea  $\{x_i, x_j\}$  și se notează cu  $[x_i, x_j]$ .  $[x_i, x_j]$  și  $[x_j, x_i]$  reprezintă aceeași muchie a grafului. Graful  $G$  are  $m$  muchii:

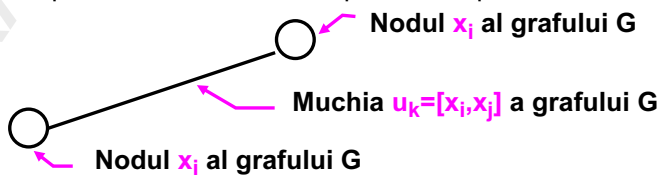
$$\text{numărul de muchii} = \text{card}(U) = m$$

- Numim **noduri adiacente** orice pereche de noduri care formează o muchie –  $\{x_i, x_j\} \in U$ . Fiecare dintre cele două noduri ( $x_i$  și  $x_j$ ) este **nod incident** cu muchia  $u_k = [x_i, x_j]$ .
- **Nodurile vecine** unui nod  $x_i$  sunt toate nodurile  $x_j$  care sunt adiacente cu el.
- Se numește **nod extrem** al unei muchii oricare dintre cele două noduri care se găsesc la capătul muchiei. Nodurile  $x_i$  și  $x_j$  sunt extremitățile muchiei  $[x_i, x_j]$ .
- Se numesc **muchii incidente** două muchii  $u_i$  și  $u_j$  care au o extremitate comună – nodul  $x_k$ .

**Un graf neorientat  $G$  este definit de o pereche de mulțimi:**

**mulțimea nodurilor sale –  $X$  și mulțimea muchiilor sale –  $U$ . El poate fi considerat ca o mulțime de noduri din care unele pot fi unite două câte două printr-o muchie.**

Graful neorientat se **reprezintă în plan** prin intermediul unor elemente geometrice: **nodurile** se reprezintă prin cercuri, iar **muchiiile** prin linii drepte care unesc anumite cercuri.



Elementele mulțimii  $X$  (nodurile) se identifică cu ajutorul unor **etichete**, care pot fi numere sau litere. Pentru simplificare, vom folosi ca etichete un șir de numere consecutive, începând cu numărul 1. De exemplu, pentru un graf cu  $n$  noduri, vom folosi etichetele: 1, 2,

3, ..., n-1, n. O muchie se va nota cu  $[i,j]$ , unde  $i$  și  $j$  sunt etichetele nodurilor incidente cu muchia. De exemplu, muchia  $[2,3]$  este muchia care unește nodurile cu etichetele 2 și 3.

### Exemplul 1:

În graful  $G_1=(X_1, U_1)$  din figura 1:

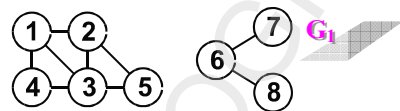


Fig. 1

- **Ordinul grafului** este 8. Graful are 8 noduri ( $n=8$ ) și **mulțimea nodurilor** este  $X_1=\{1,2,3,4,5,6,7,8\}$ .
- Graful are 9 muchii ( $m=9$ ) și **mulțimea muchiilor** este  $U_1=\{[1,2], [1,3], [1,4], [2,3], [2,5], [3,4], [3,5], [6,7], [6,8]\}$ .
- Nodul 1 este **nod adiacent** cu nodurile 2, 3 și 4, iar nodul 6 este adiacent cu nodurile 7 și 8. Nodurile 3 și 4 sunt adiacente deoarece perechea de noduri  $[3,4] \in U_1$ . Nodurile 5 și 6 nu sunt adiacente deoarece perechea de noduri  $[5,6] \notin U_1$ .
- Nodul 5 este **nod incident** cu muchiile  $[2,5]$  și  $[3,5]$ , dar nu este incident cu muchia  $[1,2]$ .
- Nodul 3 este **nod extrem** muchiilor  $[1,3]$ ,  $[2,3]$ ,  $[3,4]$  și  $[3,5]$ .
- Muchiile  $[1,2]$  și  $[2,3]$  sunt **muchii incidente** deoarece au un nod comun (nodul 2). Muchiile  $[1,4]$  și  $[2,3]$  nu sunt muchii incidente deoarece nu au un nod comun.

### Teorema 1

Dacă graful neorientat  $G$  are  $n$  noduri ( $x_1, x_2, \dots, x_n$ ), atunci **numărul total de grafuri neorientate care se pot forma cu aceste noduri este g:**

$$g = 2^{C_n^2}$$

**Demonstrație.** Notăm cu  $X$  mulțimea nodurilor grafului, cu  $U$  mulțimea muchiilor, cu  $A$  mulțimea tuturor submulțimilor de două elemente din  $X$  și cu  $B$  mulțimea  $\{0,1\}$ . Mulțimea are următoarele elemente (submulțimi):

|          |          |          |     |           |                  |
|----------|----------|----------|-----|-----------|------------------|
| $[1,2],$ | $[1,3],$ | $[1,4],$ | ... | $[1,n]$   | $n-1$ submulțimi |
|          | $[2,3],$ | $[2,4],$ | ... | $[2,n]$   | $n-2$ submulțimi |
| .....    |          |          |     |           |                  |
|          |          |          |     | $[n-1,n]$ | 1 submulțime     |

Numărul total de submulțimi este:  $(n-1) + (n-2) + \dots + 1 = \frac{n \times (n-1)}{2} = C_n^2$

Notăm cu  $a = \text{card}(A)$  și cu  $b = \text{card}(B)$ . Fiecărui graf îi putem asocia o funcție  $f: A \rightarrow B$  definită alăturat. Invers, unei funcții  $f: A \rightarrow B$  îi putem atașa un graf, astfel:  $f(\{x,y\})=1$  dacă și numai dacă  $[x,y] \in U$ . Rezultă că numărul total de grafuri care se pot forma cu  $n$  noduri este egal cu numărul de funcții  $f$ .

$$f(\{x,y\}) = \begin{cases} 1, & \text{dacă } [x,y] \in U \\ 0, & \text{dacă } [x,y] \notin U \end{cases}$$

Dar, numărul de funcții  $f: A \rightarrow B$  este egal cu  $b^a$ , unde  $b=2$  și  $a = C_n^2$

### Temă



1. În graful  $G_1$  – cu ce noduri este adiacent nodul 1?
2. În graful  $G_1$  – cu ce muchii este incident nodul 1?
3. Dați exemple de două noduri adiacente și de două noduri care nu sunt adiacente în graful  $G_1$ .
4. Dați exemple de două muchii incidente și de două muchii care nu sunt incidente în graful  $G_1$ .
5. Desenați graful  $G_2=(X_2, U_2)$  definit astfel:  
 $X_2=\{1,2,3,4,5,6,7,8\}$   
 $U_2=\{[1,2], [1,3], [1,5], [2,3], [2,5], [3,4], [4,5], [4,6], [4,7]\}$ .
6. Desenați graful traseelor rutiere care fac legătura între localitățile Brașov, București, Buzău, Ploiești și Constanța. Dacă există mai multe trasee rutiere între două localități

(de exemplu, București și Brașov), adăugați la graf noduri pentru localitățile care identifică unic aceste trasee (de exemplu, Vălenii de Munte, Târgoviște și Pitești).

7. Desenați grafurile județelor din România (între două județe există o muchie dacă cele două județe sunt învecinate).
8. Câte grafuri se pot construi cu 3 muchii? Desenați toate grafurile care se pot construi cu 3 muchii.
9. Pentru grafurile  $G_3$  din figura 2, precizați ordinul, numărul de noduri, numărul de muchii și mulțimile  $X_3$  și  $U_3$ .
10. Structura unei molecule de substanță chimică poate fi reprezentată printr-un graf neorientat, în care nodurile sunt atomii și grupările din care este compusă molecula, iar muchiile sunt legăturile dintre ele. În figura 3 este prezentat grafurile moleculei de apă  $H_2O$ . Reprezentați grafurile moleculelor de  $H_2SO_4$ ,  $NH_3$ ,  $CH_4$  și  $C_2H_4$ .

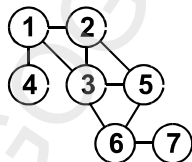


Fig. 2

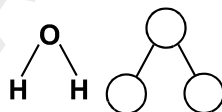


Fig. 3

### 2.6.2.2. Gradul unui nod al grafului neorientat

Nodul unui graf este caracterizat prin **grad**.

**Gradul unui nod  $x_k$  al grafului  $G$  este egal cu numărul muchiilor incidente cu nodul și se notează cu  $d(x_k)$ .**

#### Terminologie:

- Se numește **nod terminal** un nod care are gradul egal cu 1 –  $d(x_k) = 1$  (este incident cu o singură muchie).
- Se numește **nod izolat** un nod care are gradul egal cu 0 –  $d(x_k) = 0$  (nu este adiacent cu nici un alt nod al grafului, adică nu se găsește în extremitatea nici unei muchii).

**Observație:** Într-un graf cu  $n$  noduri, oricare ar fi nodul  $x_k$ , gradul său este mai mare sau egal cu 0 și mai mic sau egal cu  $n-1$  ( $0 \leq d(x_k) \leq n-1$ ).

#### Exemplul 1:

Graful  $G_4 = (X_4, U_4)$  din figura 4 este definit astfel:

$$X_4 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

$$U_4 = \{[1, 2], [1, 4], [2, 3], [2, 5], [3, 4], [3, 5], [5, 6], [5, 7], [5, 8], [7, 9]\}$$

În grafurile  $G_4$ :

- **Gradul** nodului 5 este 4, deoarece are 4 muchii incidente:  $[3, 5]$ ,  $[5, 6]$ ,  $[5, 7]$  și  $[5, 8]$ .
- Nodul 9 este **nod terminal**, deoarece are gradul 1 (o singură muchie incidentă:  $[7, 9]$ ).
- Nodul 10 este **nod izolat**, deoarece are gradul 0 (nici o muchie incidentă).

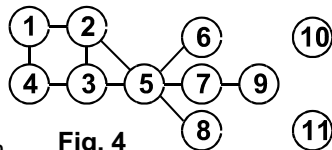


Fig. 4

#### Exemplul 2:

Fie grafurile  $G_5 = (X_5, U_5)$ , unde  $X_5 = \{1, 2, 3, 4, 5, 6, 7, 8\}$  și  $U_5 = \{[1, 2], [1, 5], [2, 3], [2, 5], [3, 4], [3, 5], [4, 5], [4, 6], [4, 7]\}$ . Din lista muchiilor unui graf neorientat, puteți preciza următoarele informații:

- Determinați **gradul** unui nod – numărând de câte ori apare eticheta nodului în lista de muchii. Nodul 5 are gradul 3 (în mulțimea muchiilor, eticheta 5 apare de 3 ori:  $[1, 5]$ ,  $[2, 5]$ ,  $[3, 5]$ ).
- Determinați dacă un nod este terminal – verificând dacă eticheta lui apare o singură dată. Nodul 7 este **nod terminal** (eticheta lui apare numai într-o singură muchie:  $[4, 7]$ ).
- Determinați dacă un nod este izolat – verificând dacă eticheta lui nu apare în lista de muchii. Nodul 8 este **nod izolat** (eticheta lui nu apare în lista muchiilor).

→ Determinați **numărul de noduri izolate** ( $n_1$ ) astfel: numărați etichetele distincte care apar în lista muchiilor ( $n_2$ ) și  $n_1 = n - n_2$ . În graful  $G_5$ , în lista de muchii, există 7 etichete distincte. Numărul de noduri izolate este 1 (8-7).

**Temă**



1. În graful  $G_4$ : precizați gradul nodului 3 și identificați nodul cu gradul cel mai mare, nodurile izolate și nodurile terminale.
2. În graful  $G_3$ : identificați nodurile care au gradul 2, precizați câte noduri au gradul impar și care este nodul cu gradul cel mai mare.
3. În graful  $G_5$ : precizați gradul nodului 3, identificați nodurile izolate și nodurile terminale, precizați câte noduri au gradul 2 și câte noduri au gradul impar.

## Teorema 2

Dacă graful  $G$  are  $m$  muchii ( $u_1, u_2, \dots, u_m$ ) și  $n$  noduri ( $x_1, x_2, \dots, x_n$ ), atunci între gradul nodurilor și numărul de muchii există următoarea relație: **suma gradelor tuturor nodurilor grafului este egală cu dublul numărului de muchii**:

$$\sum_{i=1}^n d(x_i) = 2m$$

**Demonstrație.** Fiecare muchie  $u_k = [x_i, x_j]$  corespunde unei unități din gradul nodului  $x_i$  și unei unități din gradul nodului  $x_j$ . Rezultă că fiecare muchie contribuie cu 2 unități la suma gradelor.

**Exemplu.** În graful  $G_4$ :  $d(1) + d(2) + d(3) + d(4) + d(5) + d(6) + d(7) + d(8) + d(9) + d(10) + d(11) = 2+2+3+2+4+1+2+1+1+0+0 = 18 = 2 \times 9 = 2 \times m$

**Propoziția 1.** Pentru orice graf  $G$ , **numărul nodurilor de grad impar este par**.

**Demonstrație.** Suma gradelor nodurilor fiind un număr par, această sumă trebuie să conțină un număr par de termeni care sunt numere impare.

**Exemplu.** În graful  $G_3$  există 4 noduri cu grad impar (3, 6, 8 și 9).

**Propoziția 2.** Numărul minim de muchii,  $m_{\min}$ , pe care trebuie să le aibă un graf neorientat, cu  $n$  noduri, ca **să nu existe noduri izolate**, este:

$$m_{\min} = \left\lceil \frac{n+1}{2} \right\rceil$$

**Demonstrație.** Pentru ca un nod  $x_i$  să nu fie izolat, trebuie ca  $d(x_i) \geq 1$ . Pentru ca toate nodurile să nu fie izolate, trebuie ca suma gradelor să fie mai mare sau egală cu  $n$ . Dar, suma gradelor este dublul numărului de muchii –  $m$ . Înseamnă că, pentru  $n$  par –  $m_{\min} = n/2$ , iar pentru  $n$  impar –  $m_{\min} = (n+1)/2$ .

## Teorema 3

Dacă graful  $G$  are  $n$  noduri ( $n \geq 2$ ), atunci **cel puțin două noduri au același grad**.

**Demonstrație** – prin reducere la absurd. Presupunem că nu este adevărat. Cum, oricare ar fi nodul  $x_k$ ,  $0 \leq d(x_k) \leq n-1$ , înseamnă că singurul șir de  $n$  numere, diferite între ele două câte două, care pot reprezenta gradele unghiurilor este  $0, 1, 2, \dots, n-1$ . Deoarece un nod este izolat, cel mai mare grad al unui nod nu poate fi decât  $n-2$  (nodul nu se poate lega de el însuși și de nodul izolat). Rezultă că șirul de numere definit mai sus (singurul șir care se poate defini) nu poate reprezenta șirul gradelor în graf.

**Temă**



1. În graful  $G_1$  – verificați că este îndeplinită relația dintre gradele nodurilor și numărul de muchii ale grafului. Identificați nodurile cu grad impar și verificați că numărul lor este par.
2. Dacă un graf are 8 noduri, care este numărul minim de muchii pe care trebuie să le aibă, ca să nu fie noduri izolate. Desenați un graf care, având numărul minim de muchii

stabilit, nu conține noduri izolate. Dacă un graf are 9 noduri, care este numărul minim de muchii pe care trebuie să le aibă, ca să nu fie noduri izolate. Desenați un graf care având numărul minim de muchii stabilit nu conține noduri izolate.

## 2.6.3. Graful orientat

Spre deosebire de graful neorientat, în graful orientat **perechile de noduri sunt ordonate**.

### 2.6.3.1. Terminologie

→ Elementele mulțimii  $U$  (perechile de noduri) se numesc **arce**. Mulțimea  $U$  se mai numește și **mulțimea arcelor** grafului  $G$ . Un arc, fiind un element din mulțimea  $U$ , este determinat de o submulțime ordonată, cu două elemente, din mulțimea  $X$ : arcul  $k$  al grafului ( $u_k$ ), ce unește nodurile  $x_i$  și  $x_j$ , este determinat de submulțimea  $\{x_i, x_j\}$  și se notează cu  $[x_i, x_j]$ .  $[x_i, x_j]$  și  $[x_j, x_i]$  **nu reprezintă același arc al grafului**. Graful  $G$  are  $m$  arce:

$$\text{numărul de arce} = \text{card}(U) = m$$

→ Se numesc **noduri adiacente** în graful  $G$  oricare din perechile de noduri care formează un arc –  $(x_i, x_j) \in U$  sau  $(x_j, x_i) \in U$ . Fiecare dintre cele două noduri ( $x_i$  și  $x_j$ ) este **nod incident** cu arcul  $u_k = [x_i, x_j]$  sau cu arcul  $u_k = [x_j, x_i]$ .

→ Nodurile  $x_i$  și  $x_j$  sunt extremitățile arcului  $[x_i, x_j]$ . Nodul  $x_i$  este **extremitatea inițială** a arcului, iar nodul  $x_j$  este **extremitatea finală** a arcului.

→ Se numesc **arce incidente** două arce  $u_i$  și  $u_j$  care au o extremitate comună – nodul  $x_k$ .

→ Se numește **succesor** al nodului  $x_i$  orice nod la care ajunge un arc care iese din nodul  $x_i$ . **Mulțimea succesorilor** nodului  $x_i$  este formată din mulțimea nodurilor la care ajung arcele care ies din nodul  $x_i$ . Se notează cu  $S(x_i)$  și se definește ca mulțimea:

$$S(x) = \{x_j \in X \mid (x_i, x_j) \in U\}.$$

→ Se numește **predecesor** al nodului  $x_i$  orice nod de la care intră un arc în nodul  $x_i$ . **Mulțimea predecesorilor** nodului  $x_i$  este formată din mulțimea nodurilor de la care ajung arcele care intră în nodul  $x_i$ . Se notează cu  $P(x_i)$  și se definește ca mulțimea:

$$P(x) = \{x_j \in X \mid (x_j, x_i) \in U\}.$$

→ **Mulțimea arcelor care ies** din nodul  $x_i$  se notează cu  $U^+(x_i)$  și se definește ca mulțimea  $U^+(x_i) = \{u = (x_i, x_j) \mid u \in U\}$ .

→ **Mulțimea arcelor care intră** în nodul  $x_i$  se notează cu  $U^-(x_i)$  și se definește ca mulțimea  $U^-(x_i) = \{u = (x_j, x_i) \mid u \in U\}$ .

→ **Nodul sursă** al grafului este nodul care are mulțimea succesorilor formată din toate celelalte noduri, mai puțin el, iar mulțimea predecesorilor săi este mulțimea vidă.

→ **Nodul destinație** al grafului este nodul care are mulțimea predecesorilor formată din toate celelalte noduri, mai puțin el, iar mulțimea succesorilor săi este mulțimea vidă.

#### Observații

1.  $\text{card}(S(x)) = \text{card}(U^+(x))$  și  $\text{card}(P(x)) = \text{card}(U^-(x))$ .
2. Pentru nodul sursă al grafului  $\text{card}(S(x)) = \text{card}(X) - 1$  și  $\text{card}(P(x)) = 0$ .
3. Pentru nodul destinație al grafului  $\text{card}(P(x)) = \text{card}(X) - 1$  și  $\text{card}(S(x)) = 0$ .
4. Dacă un graf are un nod sursă, atunci nu poate avea un nod destinație, și invers.

**Un graf orientat  $G$  este definit de o pereche de mulțimi: mulțimea nodurilor sale –  $X$  și mulțimea arcelor sale –  $U$ . El poate fi considerat ca o mulțime de noduri din care unele pot fi unite două câte două, prin unul sau două arce.**

Graful orientat se **reprezintă în plan** prin intermediul unor elemente geometrice: **nodurile** se reprezintă prin cercuri, iar **arcele** prin linii drepte care unesc anumite cercuri și care au o săgeată la capătul care corespunde extremității finale a arcului.

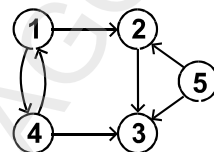
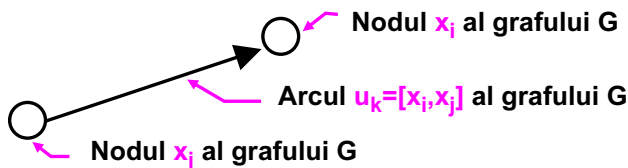


Fig. 5

### Exemplu:

În graful  $G_6=(X_6, U_6)$  – din figura 5:

- **Ordinul grafului** este 5.
- Graful are 5 noduri ( $n=5$ ) și **mulțimea nodurilor** este  $X_6=\{1,2,3,4,5\}$ .
- Graful are 7 arce ( $m=7$ ) și **mulțimea arcelor** este  $U_6=\{[1,2], [1,4], [2,3], [4,1], [4,3], [5,2], [5,3]\}$ .
- Nodul 1 este **nod adiacent** cu nodurile 2 și 4, iar nodul 3 este adiacent cu nodurile 2, 4 și 5. Nodurile 3 și 4 sunt adiacente deoarece perechea de noduri  $[4,3] \in U$ . Nodurile 5 și 4 nu sunt adiacente, deoarece nici una dintre perechile de noduri  $[4,5]$  și  $[5,4] \notin U$ .
- Nodul 4 este **nod incident** cu arcele  $[1,4]$ ,  $[4,1]$  și  $[4,3]$ , dar nu este incident cu arcul  $[1,2]$ .
- Nodul 2 este **extremitatea inițială** a arcului  $[2,3]$  și **extremitatea finală** a arcelor  $[1,2]$  și  $[5,2]$ .
- Arcele  $[1,2]$  și  $[5,2]$  sunt **arce incidente** deoarece au un nod comun (nodul 2). Arcele  $[1,4]$  și  $[2,3]$  nu sunt arce incidente, deoarece nu au un nod comun.
- **Mulțimea succesorilor** nodului 1 este formată din nodurile 2 și 4. Nodul 2 este **nod succesor** al nodului 1, dar și al nodului 5. Nodul 1 este nod succesor al nodului 4, dar și nodul 4 este nod succesor al nodului 1. Nodul 3 nu are succesori.
- **Mulțimea predecesorilor** nodului 3 este formată din nodurile 2, 4 și 5. Nodul 2 este **nod predecesor** al nodului 3. Nodul 1 este nod predecesor al nodului 4, dar și nodul 4 este nod predecesor al nodului 1. Nodul 5 nu are predecesori.

### Teorema 4

Dacă graful orientat  $G$  are  $n$  noduri ( $x_1, x_2, \dots, x_n$ ), atunci **numărul total de grafuri orientate care se pot forma cu aceste noduri este g:**

$$g = 4^{\frac{n \times (n-1)}{2}}$$

**Demonstrație.** Se demonstrează la fel ca **Teorema 1** cu deosebirea că mulțimea  $B$  este  $\{0,1,2,3\}$  –  $\text{card}(B)=4$ , iar funcția  $f$  este definită alăturat.

$$f(\{x,y\}) = \begin{cases} 3, & \text{dacă } [x,y] \in U \text{ și } [y,x] \in U \\ 2, & \text{dacă } [x,y] \in U \\ 1, & \text{dacă } [y,x] \in U \\ 0, & \text{dacă } [x,y] \notin U \text{ și } [y,x] \notin U \end{cases}$$

### Temă



1. În graful  $G_6$  – cu ce noduri este adiacent nodul 2?
2. În graful  $G_6$  – cu ce arce este incident nodul 2?
3. Dați exemplu de două noduri adiacente în graful  $G_6$ .
4. Dați exemplu de două noduri care nu sunt adiacente în graful  $G_6$ .
5. Dați exemplu de două arce incidente în graful  $G_6$ .
6. În graful  $G_6$  precizați ce fel de extremitate este nodul 4 pentru fiecare arc cu care este incident. Precizați mulțimea succesorilor și mulțimea predecesorilor nodului 4.

- G<sub>7</sub>** 7. Desenați graful  $G_7=(X_8, U_8)$ , definit astfel.  
 $X_7=\{1,2,3,4,5,6,7\}$   
 $U_7=\{[1,2], [1,5], [2,1], [2,4], [3,4], [4,3], [5,3], [6,5], [6,7], [7,5]\}$ .
8. Câte grafuri orientate se pot construi cu 3 arce? Desenați 10 dintre aceste grafuri.
- G<sub>8</sub>** 9. Pentru graful  $G_8$  din figura 6, precizați ordinul, numărul de noduri, numărul de arce și mulțimile  $X_8$  și  $U_8$ .

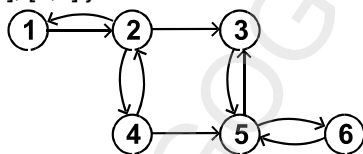


Fig. 6

### 2.6.3.2. Gradele unui nod al grafului orientat

Nodul unui graf orientat este caracterizat prin **gradul intern** și **gradul extern**.

**Gradul intern al unui nod  $x_i$**  al grafului  $G$  este egal cu numărul arcelor care intră în nodul  $x_i$  (arce de forma  $[x_j, x_i]$ ) și se notează cu  $d^-(x_i)$ .

**Gradul extern al unui nod  $x_i$**  al grafului  $G$  este egal cu numărul arcelor care ies din nodul  $x_i$  (arce de forma  $[x_i, x_j]$ ) și se notează cu  $d^+(x_i)$ .

#### Terminologie:

- Se numește **nod terminal** un nod care are suma gradelor egală cu 1 (gradul intern sau gradul extern egal cu 1 și gradul extern, respectiv gradul intern, egal cu 0 –  $d^+(x_k) = 1$  și  $d^-(x_k) = 0$  sau  $d^-(x_k) = 1$  și  $d^+(x_k) = 0$ ). Nodul terminal este incident cu un singur arc.
- Se numește **nod izolat** un nod care are suma gradelor egală cu 0 (gradul intern și gradul extern egale cu 0 –  $d^+(x_k) = d^-(x_k) = 0$ ). Nodul izolat nu este adiacent cu nici un alt nod al grafului, adică nu se găsește la extremitatea niciunui arc.

#### Observații:

1.  $d^+(x) = \text{card}(S(x))$  și  $d^-(x) = \text{card}(P(x))$ .
2. Dacă graful are  $n$  noduri, pentru un nod sursă al grafului  $d^+(x) = n-1$  și  $d^-(x) = 0$ , iar pentru un nod destinație al grafului  $d^-(x) = n-1$  și  $d^+(x) = 0$ .

**Observație:** Într-un graf cu  $n$  noduri, oricare ar fi nodul  $x_k$ , oricare dintre gradele sale este mai mare sau egal cu 0 și mai mic sau egal cu  $n-1$  ( $0 \leq d^+(x_k) \leq n-1$  și  $0 \leq d^-(x_k) \leq n-1$ ).

#### Exemplul 1:

Graful  $G_9=(X_9, U_9)$  din figura 7 este definit astfel:

$$X_9=\{1,2,3,4,5,6,7,8,9,10\}$$

$$U_9=\{[1,2], [1,4], [2,1], [2,3], [2,5], [2,6], [2,7], [4,1], [7,2], [8,9], [9,8]\}$$

În graful  $G_9$ :

- **Gradul intern** al nodului 2 este 2, deoarece are 2 arce care intră:  $[1,2]$  și  $[7,2]$ . **Gradul extern** al nodului 2 este 4, deoarece are 4 arce care ies:  $[2,1]$ ,  $[2,3]$ ,  $[2,5]$  și  $[2,7]$ .
- Nodul 5 este **nod terminal** deoarece are suma gradelor egală cu 1 (gradul intern este 1 și gradul extern este 0) și un singur arc incident:  $[2,5]$ .
- Nodul 10 este **nod izolat**, deoarece are gradul 0 (niciun arc incident).

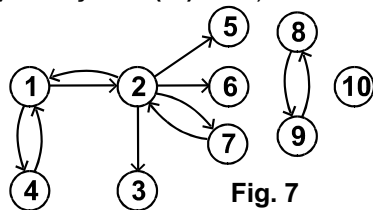


Fig. 7

#### Exemplul 2:

Fie graful  $G_{10}=(X_{10}, U_{10})$ , unde  $X_{10}=\{1,2,3,4,5,6,7,8\}$  și  $U_{10}=\{[1,2], [1,5], [2,1], [2,3], [2,5], [3,4], [3,5], [4,3], [4,5], [4,6], [4,7], [5,4]\}$ . Din lista arcelor unui graf orientat, puteți preciza următoarele informații:



- **Gradul extern** al unui nod – numărând de câte ori apare eticheta nodului în lista de arce, ca prim element din pereche. Nodul **3** are gradul extern 2 (în mulțimea arcelor, eticheta 3 apare de 2 ori ca prim element:  $[3,4]$  și  $[3,5]$ ).
- **Gradul intern** al unui nod – numărând de câte ori apare eticheta nodului în lista de arce, ca al doilea element din pereche. Nodul **5** are gradul 4 (în mulțimea arcelor, eticheta 5 apare de 4 ori ca al doilea element:  $[1,5]$ ,  $[2,5]$ ,  $[3,5]$  și  $[4,5]$ ).
- **Mulțimea succesorilor unui nod** este formată din nodurile a căror etichetă apare ca al doilea element în perechile în care primul element este nodul precizat. Mulțimea succesorilor nodului **4** este  $\{3, 5, 6, 7\}$  – arcele:  $[4,3]$ ,  $[4,5]$ ,  $[4,6]$  și  $[4,7]$ .
- **Mulțimea predecesorilor unui nod** este formată din nodurile a căror etichetă apare ca prim element în perechile în care al doilea element este nodul precizat. Mulțimea predecesorilor nodului **3** este  $\{2, 4\}$  – arcele:  $[2,3]$  și  $[4,3]$ .

### Exemplul 3

Fie grafurile  $G_{11}=(X_{11}, U_{11})$ , unde  $X_{11}=\{1,2,3,4\}$  și  $U_{11}=\{[1,2], [1,3], [1,4], [2,3], [3,4], [4,3]\}$ , și  $G_{12}=(X_{12}, U_{12})$ , unde  $X_{12}=\{1,2,3,4\}$  și  $U_{12}=\{[2,1], [2,3], [3,1], [3,4], [4,1], [4,3]\}$ . Din lista muchiilor unui graf, puteți preciza următoarele informații:

- **Nodul sursă** al unui graf apare pe primul loc din pereche de **n-1** ori – și niciodată pe locul al doilea. În graful  $G_{11}$ , nodul 1 este nod sursă. Desenați graful  $G_{11}$  pentru a verifica această afirmație.
- **Nodul destinație** al unui graf apare pe al doilea loc din pereche de **n-1** ori – și niciodată pe primul loc. În graful  $G_{12}$ , nodul 1 este nod destinație. Desenați graful  $G_{12}$  pentru a verifica această afirmație.

### Temă



1. În graful  $G_8$  – precizați gradul intern și gradul extern ale nodului 5, identificați nodul cu gradul extern cel mai mare și nodurile cu gradul intern cel mai mic.
2. În graful  $G_9$  – identificați nodurile care au gradul intern 1, precizați câte noduri au gradul intern egal cu gradul extern, care sunt nodurile terminale și care sunt nodurile izolate.
3. În graful  $G_{10}$  – precizați gradul intern și gradul extern ale nodului 4, identificați nodurile izolate și nodurile terminale, identificați nodurile care au gradul extern maxim și nodurile care au gradul intern egal cu gradul extern.

### Teorema 5

Dacă graful orientat  $G$  are  **$m$  arce** ( $u_1, u_2, \dots, u_m$ ) și  **$n$  noduri** ( $x_1, x_2, \dots, x_n$ ), atunci între gradele nodurilor și numărul de muchii există următoarea relație: **suma gradelor interne ale tuturor nodurilor este egală cu suma gradelor externe ale tuturor nodurilor și cu numărul de arce:**

$$\sum_{i=1}^n d^+(x_i) = \sum_{i=1}^n d^-(x_i) = m$$

**Demonstrație.** Fiecare arc  $u_k = [x_i, x_j]$  corespunde unei unități din gradul extern al nodului  $x_i$  și unei unități din gradul intern al nodului  $x_j$ . Rezultă că fiecare arc contribuie cu o unitate la suma gradelor interne și cu o unitate la suma gradelor externe.

**Exemplu.** În graful  $G_8$ :  $d^+(1) + d^+(2) + d^+(3) + d^+(4) + d^+(5) + d^+(6) = 1+3+1+2+2+1 = 10$  și  $d^-(1) + d^-(2) + d^-(3) + d^-(4) + d^-(5) + d^-(6) = 1+2+2+2+2+1 = 10$ ,  $m$  fiind egal cu 10.

### Temă



În graful  $G_{10}$  verificați că este îndeplinită relația dintre gradurile nodurilor și numărul de arce ale grafului.

## 2.6.4. Reprezentarea și implementarea grafului

Există mai multe moduri de reprezentare la nivel logic a unui graf, care pot fi implementate în memoria unui calculator, folosind diverse tipuri de structuri de date. Aceste reprezentări pot fi folosite în algoritmi care prelucreează grafuri și, implicit, în programele prin care vor fi implementați în calculator acești algoritmi. Printre modurile de reprezentare a unui graf se numără:

- reprezentarea prin **matricea de adiacență**;
- reprezentarea prin **lista muchiilor (arcelor)**;
- reprezentarea prin **lista de adiacență (listele vecinilor)**.

Fiecare reprezentare prezintă avantaje în ceea ce privește utilizarea eficientă a memoriei interne, în funcție de tipul grafului (cu noduri puține, dar cu muchii multe – sau cu noduri multe, dar cu muchii puține) și din punct de vedere al eficienței algoritmilor de prelucrare (în funcție de aplicație). În următoarele reprezentări se consideră că graful  $G=(X,U)$  are  $n$  noduri și  $m$  muchii.

### 2.6.4.1. Reprezentarea prin matricea de adiacență

**Matricea de adiacență** a unui graf este o matrice pătrată **binară** de ordinul  $n$  ( $A_{n,n}$ ), ale cărei elemente  $a_{i,j}$  sunt definite astfel:

$$a_{i,j} = \begin{cases} 1, & \text{dacă } [i,j] \in U \\ 0, & \text{dacă } [i,j] \notin U \end{cases}$$

Implementarea grafului prin matricea de adiacență se face printr-un tablou bidimensional (o matrice pătrată cu dimensiunea  $n$ ), astfel:

```
int a [<n>] [<n>];
```

Exemple:

**Graful neorientat  $G_1$**

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

**Graful orientat  $G_8$**

|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 |

### Proprietățile matricei de adiacență:

1. Elementele de pe diagonala principală au valoarea 0 – din definiția grafului rezultă că orice muchie (arc)  $[i, j]$  trebuie să respecte condiția  $i \neq j$ .
2. În cazul unui **graf neorientat**, matricea de adiacență este o **matrice simetrică** față de diagonala principală, deoarece, dacă există muchia  $[i, j]$ , atunci există și muchia  $[j, i]$ .

**Această reprezentare este recomandată pentru problemele în care se testează prezența unei muchii sau a unui arc între două noduri, se calculează gradul unui nod etc. – deoarece permite un acces rapid la nodurile și muchiile (arcele) unui graf.**

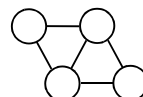
## Algoritmi pentru reprezentarea grafurilor folosind matricea de adiacență

Din matricea de adiacență puteți obține următoarele informații:

| Graf neorientat                                                                                                                                                                                                                                                                                                    | Graf orientat                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Suma elementelor matricei de adiacență este egală cu $2 \times m$ ( <b>dublul numărului de muchii</b> ).                                                                                                                                                                                                           | Suma elementelor matricei de adiacență este egală cu $m$ ( <b>numărul de arce</b> ).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Gradul unui nod <math>i</math></b> este egal cu <b>suma elementelor de pe linia <math>i</math></b> (sau cu <b>suma elementelor din coloana <math>i</math></b> ).                                                                                                                                                | <b>Gradul extern al nodului <math>i</math></b> este egal cu <b>suma elementelor de pe linia <math>i</math></b> .<br><b>Gradul intern al nodului <math>i</math></b> este egal cu <b>suma elementelor din coloana <math>i</math></b> .                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Nodurile adiacente nodului <math>i</math></b> sunt nodurile $j$ ( $j=1, n$ ) pentru care <b>elementele din linia <math>i</math> sunt egale cu 1</b> ( $a[i][j]=1$ ). Mai pot fi definite ca nodurile $j$ ( $j=1, n$ ) pentru care <b>elementele din coloana <math>i</math> sunt egale cu 1</b> ( $a[j][i]=1$ ). | <b>Succesorii nodului <math>i</math></b> sunt nodurile $j$ ( $j=1, n$ ) pentru care elementele din <b>linia <math>i</math></b> sunt egale cu 1 ( $a[i][j]=1$ ).<br><b>Predecesorii nodului <math>i</math></b> sunt nodurile $j$ ( $j=1, n$ ) pentru care elementele din <b>coloana <math>i</math></b> sunt egale cu 1 ( $a[j][i]=1$ ).<br><b>Nodurile adiacente nodului <math>i</math></b> sunt nodurile $j$ ( $j=1, n$ ) pentru care elementele din <b>linia <math>i</math> sau din coloana <math>i</math></b> sunt egale cu 1 ( $a[i][j]=1$ sau $a[j][i]=1$ ) – reuniunea dintre mulțimea succesorilor și mulțimea predecesorilor nodului. |
| <b>Numărul de vecini ai nodului <math>i</math></b> este egal cu <b>gradul nodului</b> .                                                                                                                                                                                                                            | <b>Numărul de vecini ai nodului <math>i</math></b> este egal cu <b>cardinalul mulțimii de noduri adiacente nodului <math>i</math></b> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Muchia <math>[i, j]</math></b> a grafului reprezintă un element al matricei de adiacență care îndeplinește condiția: <b><math>a[i][j] = 1</math> (sau <math>a[j][i] = 1</math>)</b> .                                                                                                                           | <b>Arcul <math>[i, j]</math></b> al grafului reprezintă un element al matricei de adiacență care îndeplinește condiția: <b><math>a[i][j] = 1</math></b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

## Exemplu

Se consideră graful din figura alăturată. Identificați matricea de adiacență a grafului.



a)

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

b)

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

c)

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |

d)

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

(Bacalaureat – Sesiunea specială 2003)

Răspunsul corect este matricea **a)**. Pentru a identifica matricea de adiacență a grafului din figură, se vor elimina pe rând variantele incorecte, prin verificarea următoarelor condiții:

- Matricea trebuie să fie **binară** – toate matricele îndeplinesc această condiție;
- Elementele de pe **diagonala principală** trebuie să aibă **valoarea 0** – matricea **b)** nu îndeplinește această condiție.
- Deoarece graful este neorientat, **matricea** trebuie să fie **simetrică** – matricea **c)** nu îndeplinește această condiție.
- Din analiza grafului se observă că două noduri au gradul 2 și două noduri au gradul 3; în matricea de adiacență trebuie să existe două linii care să conțină două elemente cu valoarea 1 și două linii care să conțină trei elemente cu valoarea 1 – matricea **d)** nu îndeplinește această condiție.

Temă



1. Scrieți matricea de adiacență a grafului  $G_4$ . Folosind informațiile din matricea de adiacență, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.

G<sub>13</sub>

2. Scrieți matricea de adiacență a grafului neorientat  $G_{13}=(X_{13},U_{13})$ , unde  $X_{13}=\{1,2,3,4\}$  și  $U_{13}=\{[1,2], [2,3], [3,4], [4,1]\}$ . Ce proprietate are acest graf?
3. Scrieți matricea de adiacență a grafului  $G_7$ . Folosind informațiile din matricea de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.

G<sub>14</sub>

4. Scrieți matricea de adiacență a grafului  $G_{10}$ . Folosind informațiile din matricea de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.

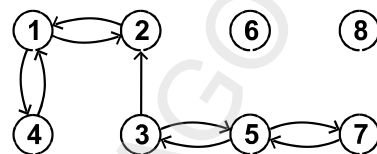


Fig. 8

5. Scrieți matricea de adiacență a grafului orientat  $G_{14}$  din figura 8.
6. Scrieți matricea de adiacență a grafului  $G_{11}$ . Cum identificați în matricea de adiacență nodul sursă al grafului?
7. Scrieți matricea de adiacență a grafului  $G_{12}$ . Cum identificați în matricea de adiacență nodul destinație al grafului?

### Implementarea algoritmilor pentru reprezentarea grafurilor cu matricea de adiacență

1. **Crearea matricei de adiacență prin introducerea datelor de la tastatură. Determinarea gradului unui nod. Salvarea matricei de adiacență într-un fișier text.**

Se citește de la tastatură muchiile (arcele) unui graf orientat (neorientat), se creează matricea de adiacență a grafului, se afișează nodurile izolate și terminale și se salvează matricea de adiacență în fișierul text *graf1.txt*, pentru graful neorientat, și *graf2.txt*, pentru graful orientat. În fișierul text se scriu, pe primul rând, ordinul grafului, și pe următoarele rânduri, liniile matricei de adiacență. Funcția `scrie()` se folosește pentru a scrie matricea de adiacență în fișier. Deoarece matricea de adiacență a fost declarată ca **variabilă globală, elementele ei sunt inițializate cu valoarea 0**. Pentru testarea programelor, se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

**Graful neorientat.** Funcția `grad()` se folosește pentru a determina gradul unui nod.

```
#include<fstream.h>
int a[10][10],n,m;
fstream f("graf1.txt",ios::out);
void scrie()
{int i,j; f<<n<<endl;
 for(i=1;i<=n;i++)
 {for (j=1;j<=n;j++) f<<a[i][j]<<" "; f<<endl;}
 f.close();}
int grad(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[i][j]; return g;}
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"primul nod al muchiei "<<k<<": "; cin>>i;
 cout<<"al doilea nod al muchiei "<<k<<": "; cin>>j;
 a[i][j]=1; a[j][i]=1;}
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==0) cout<<i<<" ";
```

```
cout<<endl<<"Nodurile terminale sunt: ";
for(i=1;i<=n;i++) if (grad(i)==1) cout<<i<<" ";
scrie();}
```

**Graful orientat.** Funcția `grad_int()` se folosește pentru a determina gradul intern al unui nod, iar funcția `grad_ext()` se folosește pentru a determina gradul extern al unui nod.

```
#include<fstream.h>
int a[10][10],n,m;
fstream f("graf2.txt",ios::out);
void scrie() { //este identică cu cea de la graful neorientat}
int grad_ext(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[i][j]; return g;}
int grad_int(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[j][i]; return g;}
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"nodul initial al arcului "<<k<<": "; cin>>i;
 cout<<"nodul final al arcului "<<k<<": "; cin>>j;
 a[i][j]=1;}
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==0) cout<<i<<" ";
 cout<<endl<<"Nodurile terminale sunt: ";
 for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==1) cout<<i<<" ";
 scrie();}
```

## 2. Crearea matricei de adiacență prin citirea datelor din fișier. Determinarea numărului de vecini ai unui nod. Afișarea muchiilor (arcelor) grafului.

Se citesc din fișierele text create anterior matricele de adiacență ale celor două grafuri – *graf1.txt*, respectiv *graf2.txt*. Se afișează nodurile care au cei mai mulți vecini (cele mai multe noduri adiacente). Se determină numărul de muchii (arce) ale grafului și se afișează muchiile (arcele). Funcția `citeste()` se folosește pentru a citi matricea de adiacență din fișier.

**Graful neorientat.** Funcția `nr_m()` se folosește pentru a determina numărul de muchii ale grafului. La afișarea muchiilor, pentru a nu se afișa de două ori aceeași muchie, **se parcurge matricea de adiacență numai deasupra diagonalei principale.**

```
#include<fstream.h>
int a[10][10],n;
fstream f("graf1.txt",ios::in);
void citeste()
{int i,j; f>>n;
 for(i=1;i<=n;i++)
 for (j=1;j<=n;j++) f>>a[i][j]; f.close();}
int nr_m()
{int i,j,m=0;
 for(i=1;i<=n;i++)
 for (j=1;j<=n;j++) m+=a[i][j];
 return m/2;}
int grad(int i){ // este identică cu cea de la exemplul anterior }
```

```

void main()
{int i,j,max; citeste(); max=grad(1);
 for(i=2;i<=n;i++) if (grad(i)>max) max=grad(i);
 cout<<"Nodurile cu cei mai multi vecini sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==max) cout<<i<<" ";
 cout<<endl<<"Graful are "<<nr_m()<<" muchii "<<endl;
 cout<<"Muchiile grafului sunt: ";
 for(i=1;i<=n;i++)
 for (j=i+1;j<=n;j++) if (a[i][j]==1) cout<<i<<"-"<<j<<" "; }

```

**Graful orientat.** Funcția `nr_a()` se folosește pentru a determina numărul de arce ale grafului. Funcția `vecini()` se folosește pentru a determina numărul de vecini ai unui nod. La afișarea arcelor, se parcurge toată matricea de adiacență.

```

#include<fstream.h>
int a[10][10],n;
fstream f("graf2.txt",ios::in);
void citeste() {//este identică cu cea de la graful neorientat}
int nr_a()
{int i,j,m=0;
 for(i=1;i<=n;i++)
 for (j=1;j<=n;j++) m+=a[i][j];
 return m;}
int vecini(int i)
{int j,v=0;
 for (j=1;j<=n;j++) if (a[i][j]==1 || a[j][i]==1) v++;
 return v;}
void main()
{int i,j,max; citeste(); max=vecini(1);
 for(i=2;i<=n;i++) if (vecini(i)>max) max=vecini(i);
 cout<<"Nodurile cu cei mai multi vecini sunt: ";
 for(i=1;i<=n;i++) if (vecini(i)==max) cout<<i<<" ";
 cout<<endl<<"Graful are "<<nr_a()<<" arce "<<endl;
 cout<<"Arcele grafului sunt: ";
 for(i=1;i<=n;i++)
 for (j=1;j<=n;j++) if (a[i][j]==1) cout<<i<<"-"<<j<<" ";}

```

### 3. Crearea matricei de adiacență prin citirea muchiilor (arcelor) din fișier. Determinarea vecinilor unui nod.

Datele se citesc din fișierul text *graf3.txt*, pentru graful neorientat, și *graf4.txt*, pentru graful orientat. În fișier sunt scrise, pe primul rând, despărțite prin spații, numărul de noduri și numărul de muchii (arce) ale grafului, și apoi pe câte un rând, separate prin spațiu, cele două noduri terminale ale fiecărei muchii (arc). Se afișează vecinii fiecărui nod. Funcția `citeste()` se folosește pentru a citi datele din fișier, iar funcția `vecini()` pentru a determina vecinii unui nod. Fișierele text se creează cu ajutorul unui editor de texte. Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

#### Graful neorientat.

```

#include<fstream.h>
int a[10][10],n,m;
fstream f("graf3.txt",ios::in);
void citeste()
{int k,i,j; f>>n>>m;

```

```

for(k=1;k<=m;k++) {f>>i>>j; a[i][j]=1; a[j][i]=1;}
f.close();}
void vecini(int i)
{for(int j=1;j<=n;j++) if(a[i][j]==1) cout<<j<<" ";}
void main()
{int i; citeste();
cout<<"Vecinii fiecarui nod sunt: "<<endl;
for (i=1;i<=n;i++) {cout<<"Nodul "<<i<<": "; vecini(i); cout<<endl;}}

```

### Graful orientat.

```

#include<fstream.h>
int a[10][10],n,m;
fstream f("graf4.txt",ios::in);
void citeste()
{int k,i,j; f>>n>>m;
for(k=1;k<=m;k++) {f>>i>>j; a[i][j]=1;}
f.close();}
void vecini(int i)
{for(int j=1;j<=n;j++) if(a[i][j]==1 || a[j][i]==1) cout<<j<<" ";}
void main()
{int i; citeste();
cout<<"Vecinii fiecarui nod sunt: "<<endl;
for (i=1;i<=n;i++){cout<<"Nodul "<<i<<": "; vecini(i); cout<<endl;}}

```

### Temă



1. Într-un fișier este scrisă o matrice pătrată, astfel: pe primul rând, un număr care reprezintă dimensiunea matricei, și pe următoarele rânduri, valori numerice despărțite prin spațiu – care reprezintă elementele de pe câte o linie a matricei. Să se verifice dacă această matrice poate fi matricea de adiacență a unui graf. În caz afirmativ, să se precizeze dacă graful este orientat sau neorientat (**Indicație.** Se verifică următoarele condiții: a) să fie o matrice binară; b) elementele de pe diagonala principală să aibă valoarea 0. Pentru graful neorientat, se verifică dacă matricea este simetrică).
2. Scrieți un program care citește dintr-un fișier matricea de adiacență a unui graf neorientat și care determină numărul minim de muchii care trebuie adăugate pentru ca graful să nu conțină noduri izolate.
3. Scrieți un program care citește, din două fișiere text, *g1.txt* și *g2.txt*, matricele de adiacență a două grafuri,  $G_a=(X,U_a)$  și  $G_b=(X,U_b)$ , și care determină matricea de adiacență a **grafului reuniune**,  $G_r=(X,U_r)$ , unde  $U_r= U_a \cup U_b$ . Matricea de adiacență se salvează în fișierul *g3.txt*.
4. Scrieți un program care citește, din două fișiere text, *g1.txt* și *g2.txt*, matricele de adiacență a două grafuri,  $G_a=(X,U_a)$  și  $G_b=(X,U_b)$ , și care determină matricea de adiacență a **grafului intersecție**  $G_i=(X,U_i)$ , unde  $U_i= U_a \cap U_b$ . Matricea de adiacență se salvează în fișierul *g4.txt*.
5. Scrieți un program care citește, din fișierul text, *graf2.txt*, informații despre un graf orientat: (de pe prima linie – numărul de noduri, apoi matricea de adiacență) și de la tastatură – o mulțime A de numere care reprezintă etichetele unor noduri din graf și care afișează mulțimea arcelor ce au o extremitate într-un nod din mulțimea A și o extremitate în mulțimea X-A (X fiind mulțimea nodurilor grafului). Pentru testarea programului, se vor folosi graful  $G_8$  și mulțimea  $A=\{1,3,4,6\}$ .



### 2.6.4.2. Reprezentarea prin lista muchiilor

**Lista muchiilor** unui graf este formată din  $m$  elemente care conțin, fiecare, câte o pereche de două noduri,  $x_i$  și  $x_j$ , care formează o muchie, adică pentru care  $[x_i, x_j] \in U$ .

Implementarea acestui tip de reprezentare se poate face folosind una dintre următoarele structuri de date:

- **Matricea muchiilor  $u$**  cu dimensiune  $m \times 2$ , în care fiecare linie corespunde unei muchii (arc) și în fiecare linie se înregistrează, în cele două coloane etichetele nodurilor care se găsesc la extremitățile muchiei (arcului).

```
int u [<m>][2];
```

Referirea la nodurile adiacente muchiei (arcului)  $i$  se face prin  $u[i][0]$  (extremitatea inițială a arcului), respectiv  $u[i][1]$  (extremitatea finală a arcului).

- **Vectorul muchiilor  $u$**  cu dimensiunea  $m$  – numărul de muchii (arce) –, ale cărei elemente sunt înregistrări, fiecare înregistrare fiind formată din două câmpuri,  $x$  și  $y$ , ce conțin etichetele nodurilor care se găsesc la extremitățile muchiei (arcului). Pentru elementele vectorului se definește tipul de dată *muchie*, de tip înregistrare.

```
struct muchie {int x,y};
muchie u [<m>];
```

Referirea la o muchie (arc)  $i$  se face prin  $u[i]$ , iar la unul dintre nodurile adiacente muchiei (arcului) se face prin  $u[i].x$  (extremitatea inițială a arcului), respectiv  $u[i].y$  (extremitatea finală a arcului).

Dacă implementarea se face folosind matricea, atunci pentru orice muchie (arc)  $i$ ,  $u[i][0] \neq u[i][1]$ . Dacă implementarea se face folosind vectorul de înregistrări, atunci pentru orice muchie (arc)  $i$ ,  $u[i].x \neq u[i].y$ .

**Această reprezentare este recomandată pentru problemele în care se face prelucrarea succesivă a muchiilor (arcelor). Are avantajul că permite adăugarea la tipul de dată *muchie* și a altor câmpuri (lungime, cost, timp etc.), corespunzător unor mărimi care pot fi asociate muchiilor (arcelor).**

**Exemple:**

**Implementarea cu matrice**

|   | 0 | 1 |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 1 | 3 |
| 3 | 1 | 4 |
| 4 | 2 | 3 |
| 5 | 2 | 4 |
| 6 | 3 | 4 |
| 7 | 3 | 5 |
| 8 | 6 | 7 |
| 9 | 6 | 8 |

**Graful neorientat  $G_1$**

**Implementarea cu vector de înregistrări**

**Muchiile**

| 1 2 3 4 5 6 7 8 9 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|-------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1                 | 2 | 1 | 3 | 2 | 4 | 2 | 3 | 2 | 4 | 3 | 4 | 3 | 5 | 6 | 7 | 6 | 8 |

## Implementarea cu matrice

|    | 0 | 1 |
|----|---|---|
| 1  | 1 | 2 |
| 2  | 2 | 1 |
| 3  | 2 | 4 |
| 4  | 3 | 2 |
| 5  | 3 | 5 |
| 6  | 4 | 2 |
| 7  | 4 | 5 |
| 8  | 5 | 3 |
| 9  | 5 | 6 |
| 10 | 6 | 5 |

Graful orientat  $G_8$ 

## Implementarea cu vector de înregistrări

Arcele

| 10 |   |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|
| 1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |   |   |   |   |   |   |   |   |   |   |
| 1  | 2 | 2 | 1 | 2 | 4 | 3 | 2 | 3 | 5  | 4 | 2 | 4 | 5 | 5 | 3 | 5 | 6 | 6 | 5 |

## Algoritmi pentru reprezentarea grafurilor folosind lista muchiilor

Din lista muchiilor puteți obține următoarele informații:

| Graf neorientat                                                                                                                                                                                                                                                                                                               | Graf orientat                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Gradul unui nod <math>i</math></b> este egal, în funcție de implementare, cu <b>numărul de apariții ale etichetei nodului</b> în matrice, respectiv în câmpurile vectorului de înregistrări.                                                                                                                               | <b>Gradul extern al nodului <math>i</math></b> este egal, în funcție de implementare, cu <b>numărul de apariții ale etichetei nodului</b> în coloana 0 a matricei, respectiv în primul câmp, în vectorul de înregistrări.<br><b>Gradul intern al nodului <math>i</math></b> este egal, în funcție de implementare, cu <b>numărul de apariții ale etichetei nodului</b> în coloana 1 a matricei, respectiv în al doilea câmp, în vectorul de înregistrări.                                                                                                                              |
| <b>Nodurile adiacente nodului <math>i</math></b> sunt, în funcție de implementare, etichetele $j$ din coloana 1, pentru care $u[k][0]=i$ , sau din coloana 0, pentru care $u[k][1]=i$ , respectiv etichetele $j$ din câmpul $u[k].y$ , pentru care $u[k].x=i$ , sau din câmpul $u[k].x$ , pentru care $u[k].y=i$ ( $k=1,m$ ). | <b>Succesorii nodului <math>i</math></b> sunt, în funcție de implementare, etichetele $j$ din coloana 1 pentru care $u[k][0]=i$ , respectiv etichetele $j$ din câmpul $u[k].y$ pentru care $u[k].x=i$ ( $k=1,m$ ).<br><b>Predecesorii nodului <math>i</math></b> sunt, în funcție de implementare, etichetele $j$ din coloana 0 pentru care $u[k][1]=i$ , respectiv etichetele $j$ din câmpul $u[k].x$ pentru care $u[k].y=i$ ( $k=1,m$ ).<br><b>Nodurile adiacente nodului <math>i</math></b> sunt date de reuniunea dintre mulțimea succesorilor și mulțimea predecesorilor nodului. |
| <b>Numărul de vecini ai nodului <math>i</math></b> este egal cu <b>gradul</b> nodului.                                                                                                                                                                                                                                        | <b>Numărul de vecini ai nodului <math>i</math></b> este egal cu cardinalul mulțimii de noduri adiacente nodului $i$ .                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

## Temă



1. Scrieți lista muchiilor a grafului  $G_4$ . Folosind informațiile din lista muchiilor, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.
2. Scrieți lista muchiilor a grafului neorientat  $G_{13}$ . Ce proprietate are acest graf?
3. Scrieți lista muchiilor a grafului  $G_{10}$ . Folosind informațiile din lista muchiilor, determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.
4. Scrieți lista muchiilor a grafului  $G_7$ . Folosind informațiile din lista muchiilor, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.

5. Scrieți lista muchiilor a grafului orientat  $G_{15}$  din figura 9. Folosind informațiile din lista muchiilor, identificați nodurile izolate.
6. Scrieți lista muchiilor a grafului  $G_{11}$ . Cum identificați, în lista muchiilor, nodul sursă?
7. Scrieți lista muchiilor a grafului  $G_{12}$ . Cum identificați, în lista muchiilor, nodul destinație?

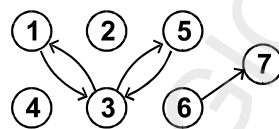


Fig. 9

### Implementarea algoritmilor pentru reprezentarea grafurilor cu lista muchiilor

1. Crearea matricei cu lista muchiilor prin introducerea datelor de la tastatură. Determinarea gradului unui nod. Salvarea informațiilor despre muchii într-un fișier text.

Se citesc de la tastatură muchiile (arcele) unui graf orientat (neorientat). Se creează matricea cu muchiile grafului. Se afișează nodurile izolate și terminale. Se salvează matricea cu muchiile grafului în fișierul text *graf11.txt*, pentru graf neorientat, și *graf12.txt*, pentru graf orientat. În fișierul text se vor scrie, pe primul rând, ordinul grafului și numărul de muchii, iar pe următoarele rânduri, nodurile de la extremitățile unei muchii (arc). Funcția `scrie()` se folosește pentru a scrie informațiile din matricea muchiilor în fișier. Pentru testarea programelor se folosesc graf neorientat  $G_1$  și graf orientat  $G_8$ .

**Graf neorientat.** Funcția `grad()` se folosește pentru a determina gradul unui nod.

```
#include<fstream.h>
int a[10][2],n,m;
fstream f("graf11.txt",ios::out);
int grad(int i)
{int k,g=0;
 for (k=1;k<=m;k++) if (a[k][0]==i || a[k][1]==i) g++; return g;}
void scrie()
{int k; f<<n<<" "<<m<<endl;
 for(k=1;k<=m;k++) f<<a[k][0]<<" "<<a[k][1]<<endl; f.close();}
void main()
{int i,j,k;
 cout<<"numar de noduri "; cin>>n; cout<<"numar de muchii "; cin>>m;
 for(k=1;k<=m;k++)
 {cout<<"primul nod al muchiei "<<k<<": "; cin>>i;
 cout<<"al doilea nod al muchiei "<<k<<": "; cin>>j;
 a[k][0]=i; a[k][1]=j;}
 cout<<"Nodurile izolate sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==0) cout<<i<<" ";
 cout<<endl<<"Nodurile terminale sunt: ";
 for(i=1;i<=n;i++) if (grad(i)==1) cout<<i<<" ";
 scrie();}
```

**Graf orientat.** Funcția `grad_int()` se folosește pentru a determina gradul intern al unui nod, iar funcția `grad_ext()` se folosește pentru a determina gradul extern al unui nod.

```
#include<fstream.h>
int a[10][2],n,m;
fstream f("graf12.txt",ios::out);
int grad_int(int i)
{int g=0,k; for(k=1;k<=m;k++) if (a[k][1]==i) g++; return g;}
int grad_ext(int i)
{int g=0,k; for(k=1;k<=m;k++) if (a[k][0]==i) g++; return g;}
void scrie() {//este identică cu cea de la graf neorientat }
```

```

void main()
{int i,j,k;
cout<<"numar de noduri "; cin>>n; cout<<"numar de arce "; cin>>m;
for (k=1;k<=m;k++)
{cout<<"nodul initial al arcului "<<k<<": "; cin>>i;
cout<<"nodul final al arcului "<<k<<": "; cin>>j;
a[k][0]=i; a[k][1]=j;}
cout<<"Nodurile izolate sunt: ";
for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==0) cout<<i<<" ";
cout<<endl<<"Nodurile terminale sunt: ";
for(i=1;i<=n;i++) if (grad_int(i)+grad_ext(i)==1) cout<<i<<" ";
scrie();}

```

## 2. Crearea vectorului de muchii prin citirea muchiilor (arcelor) din fișier. Prelucrarea informațiilor asociate muchiilor.

Datele se citesc din fișierele text create anterior: *graf7.txt*, pentru graful neorientat, și *graf8.txt*, pentru graful orientat. Pentru un nod **p** a cărui etichetă se citește de la tastatură, se afișează cel mai apropiat vecin (sau cei mai apropiați vecini, dacă există mai mulți) la care se poate ajunge din nodul **p**. În cazul grafului neorientat, cel mai apropiat vecin este nodul adiacent nodului **p** care formează cu acesta muchia care are lungimea cea mai mică față de muchiile incidente cu ceilalți vecini. În cazul grafului orientat, cel mai apropiat vecin la care se poate ajunge din nodul **p** este nodul succesor nodului **p** care formează cu acesta arcul care are lungimea cea mai mică față de arcele incidente cu ceilalți succesori. Funcția **citeste()** se folosește pentru a citi datele din fișier. Pentru testarea programelor, se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

**Graful neorientat.** Funcția **izolat()** determină dacă nodul este izolat.

```

#include<fstream.h>
struct muchie {int x,y,d;};
muchie u[20]; int n,m;
fstream f("graf7.txt",ios::in);
void citeste()
{int k; f>>n>>m;
for(k=1;k<=m;k++) f>>u[k].x>>u[k].y>>u[k].d; f.close();}
int izolat(int i)
{int k,g=0;
for (k=1;k<=m;k++) if (u[k].x==i || u[k].y==i) g++; return g==0;}
void main()
{int k,p,min;
citeste(); cout<<"Nodul: "; cin>>p;
if (izolat(p)) cout<<"Nodul "<<p<<" nu are vecini";
else
{k=1; while (u[k].x!=p && u[k].y!=p) k++;
min=u[k].d;
for (k++;k<=m;k++)
if (u[k].x==p || u[k].y==p)
if (u[k].d<min) min=u[k].d;
cout<<"Distanța minimă este "<<min<<endl;
cout<<"Nodurile aflate la distanța minimă sunt: ";
for (k=1;k<=m;k++)
{if (u[k].x==p && u[k].d==min) cout<<u[k].y<<" ";
if (u[k].y==p && u[k].d==min) cout<<u[k].x<<" ";}}}

```

**Graful orientat.** Funcția `succ()` determină dacă nodul are succesori.

```
#include<fstream.h>
struct arc {int x,y,d;};
arc u[20]; int n,m;
fstream f("graf8.txt",ios::in);
void citeste() { //este identică cu cea de la graful neorientat }
int succ(int i)
{int k,g=0; for (k=1;k<=m;k++) if (u[k].x==i) g++; return g!=0;}
void main()
{int k,p,min; citeste(); cout<<"Nodul: "; cin>>p;
if (!succ(p))
cout<<"Nodul "<<p<<" nu are vecini la care sa se poata ajunge";
else
{k=1;
while (u[k].x!=p) k++;
min=u[k].d;
for (k++;k<=m;k++) if (u[k].x==p && u[k].d<min) min=u[k].d;
cout<<"Distanța minimă este "<<min<<endl;
cout<<"Nodurile aflate la distanța minimă sunt: ";
for (k=1;k<=m;k++)
if (u[k].x==p && u[k].d==min) cout<<u[k].y<<" ";}}
```

### Temă



1. Scrieți un program care realizează următoarele:

- reface matricea cu muchiile grafului orientat din fișierul text *graf12.txt*;
- determină câte noduri izolate are graful și afișează aceste noduri

(Indicație. Se numără nodurile distincte care apar în lista muchiilor –  $n1$ , iar numărul de noduri izolate va fi dat de diferența  $n-n1$ .),

- generează matricea de adiacență din matricea muchiilor și o salvează în fișierul text *graf12a.txt*.

2. Scrieți un program care construiește vectorul de muchii din matricea de adiacență a grafului orientat care se citește din fișierul text *graf2.txt* și care determină vecinii unui nod  $p$  a cărui etichetă se citește de la tastatură.

### 2.6.4.3. Reprezentarea prin lista de adiacență

**Lista de adiacență** este formată din listele  $L_i$  ( $1 \leq i \leq n$ ) care conțin toți vecinii unui nod  $x_i$  la care se poate ajunge direct din nodul  $x_i$ , adică toate nodurile  $x_j$  pentru care  $[x_i, x_j] \in U$ .

**Observație.** În cazul grafului neorientat, lista  $L_i$  a vecinilor unui nod  $x_i$  al grafului este formată din nodurile  $x_j$  adiacente nodului  $x_i$ . În cazul grafului orientat, lista  $L_i$  a vecinilor unui nod  $x_i$  al grafului este formată din nodurile  $x_j$  care sunt succesorii nodului  $x_i$ .

Implementarea acestui tip de reprezentare se poate face folosind **matricea listei de adiacență**,  $L$  care are  $2$  linii și  $n+2 \times m$  coloane pentru graful neorientat, respectiv  $n+m$  coloane pentru graful orientat. Ea este definită astfel:

- Prima linie** conține etichetele nodurilor și listele de adiacență ale fiecărui nod; este formată din două secțiuni:

- Primele  $n$  coloane conțin etichetele nodurilor:  $L[0][i] = i$  ( $1 \leq i \leq n$ ).

**Graful neorientat  $G_1$**

| Nod | Lista de adiacență |
|-----|--------------------|
| 1   | 2, 3, 4            |
| 2   | 1, 3, 5            |
| 3   | 1, 2, 4, 5         |
| 4   | 1, 3               |
| 5   | 2, 3               |
| 6   | 7, 8               |
| 7   | 6                  |
| 8   | 6                  |

b. Următoarele  $m \times 2$  coloane, respectiv  $m$  coloane, conțin, în ordine, cele  $n$  liste de adiacență ale celor  $n$  noduri.

- **A doua linie** conține informațiile necesare pentru a identifica, în prima linie, lista de adiacență a fiecărui nod; este formată din două secțiuni:

- a. Primele  $n$  coloane conțin, în ordine, pentru fiecare nod  $i$  ( $1 \leq i \leq n$ ), indicele coloanei din prima linie din care începe lista de adiacență a nodului:  $L[1][i]=j$ , unde  $j$  este indicele coloanei de unde începe lista de adiacență a nodului  $i$ . Dacă nodul este izolat, se va memora valoarea 0 –  $L[1][i]=0$  (nu există listă de adiacență pentru acel nod).
- b. Următoarele  $m \times 2$  coloane, respectiv  $m$  coloane, conțin, în ordine, informații despre modul în care se înlănțuiesc elementele din listă. Dacă nodul  $L[0][i]$  se găsește în interiorul listei, atunci  $L[1][i]=i+1$  (indicele următorului element din listă). Dacă nodul  $L[0][i]$  se găsește la sfârșitul listei, atunci  $L[1][i]=0$  (s-a terminat lista de adiacență a nodului  $i$ ).

Graful orientat  $G_9$ 

| Nod | Listă de adiacență |
|-----|--------------------|
| 1   | 2                  |
| 2   | 1, 3, 4            |
| 3   | 5                  |
| 4   | 2, 5               |
| 5   | 3, 6               |
| 6   | 5                  |

Matricea este definită astfel:

a) pentru grafurile neorientate: `int L[2][<n>+2*<m>];`

b) pentru grafurile orientate: `int L[2][<n>+<m>];`

Graful neorientat  $G_1$ 

| Nodurile |    |    |    |    |    |    |    | L <sub>1</sub> |    |    | L <sub>2</sub> |    |    | L <sub>3</sub> |    |    |    | L <sub>4</sub> |    | L <sub>5</sub> |    | L <sub>6</sub> |    | L <sub>7</sub> | L <sub>8</sub> |
|----------|----|----|----|----|----|----|----|----------------|----|----|----------------|----|----|----------------|----|----|----|----------------|----|----------------|----|----------------|----|----------------|----------------|
| 1        | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 2              | 3  | 4  | 1              | 3  | 5  | 1              | 2  | 4  | 5  | 1              | 3  | 2              | 3  | 7              | 8  | 6              | 6              |
| 9        | 12 | 15 | 19 | 21 | 23 | 25 | 26 | 10             | 11 | 0  | 13             | 14 | 0  | 16             | 17 | 18 | 0  | 20             | 0  | 22             | 0  | 24             | 0  | 0              | 0              |
| 1        | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9              | 10 | 11 | 12             | 13 | 14 | 15             | 16 | 17 | 18 | 19             | 20 | 21             | 22 | 23             | 24 | 25             | 26             |

Graful orientat  $G_8$ 

Indicii coloanelor

| Nodurile |   |    |    |    |    | L <sub>1</sub> | L <sub>2</sub> |    | L <sub>3</sub> | L <sub>4</sub> |    | L <sub>5</sub> |    | L <sub>6</sub> |    |
|----------|---|----|----|----|----|----------------|----------------|----|----------------|----------------|----|----------------|----|----------------|----|
| 1        | 2 | 3  | 4  | 5  | 6  | 2              | 1              | 3  | 4              | 5              | 2  | 5              | 3  | 6              | 8  |
| 7        | 8 | 11 | 12 | 14 | 16 | 0              | 9              | 10 | 0              | 0              | 20 | 0              | 22 | 0              | 0  |
| 1        | 2 | 3  | 4  | 5  | 6  | 7              | 8              | 9  | 10             | 11             | 12 | 13             | 14 | 15             | 16 |

Indicii coloanelor

**Observație.** Fiecare listă de vecini  $L_i$ , conține indicii coloanelor  $j$  în care se găsesc valori de 1 în matricea de adiacență ( $a[i][j]=1$ ).

Această reprezentare este recomandată pentru grafurile care au un număr mare de noduri și un număr mic de muchii.

### Algoritmi pentru reprezentarea grafurilor folosind listele de adiacență

Din matricea listei de adiacență puteți obține următoarele informații:

| Graf neorientat                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Graf orientat                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Lungimea listei de adiacență</b> a nodului <math>i</math> neizolat, cu eticheta mai mică decât <math>n</math>, este egală cu diferența dintre primul indice <math>j</math> (<math>j=i+1, n-1</math>) diferit de 0, din prima secțiune a liniei a doua (<math>L[1][j] \neq 0</math>), și indicele coloanei din care începe lista de adiacență a nodului <math>i</math> (<math>L[1][i] - L[1][i]</math>). Pentru nodul <math>n</math>, lungimea listei de adiacență este egală cu diferența dintre numărul total de coloane, plus 1, și indicele coloanei din care începe lista de adiacență a nodului <math>n</math> (<math>n+2 \times m+1 - L[1][n]</math>). Lungimea listei de adiacență se mai poate determina prin numărarea în linia a doua a elementelor diferite de 0, începând cu elementul din coloana <math>L[1][i]</math>, la care se adaugă 1.</p> <p><b>Gradul unui nod</b> <math>i</math> este egal cu lungimea listei de adiacență a nodului sau cu numărul de apariții ale etichetei nodului în a doua secțiune a primei linii a matricei (<math>L[0][j]=i</math>, cu <math>j=n+1, n+2 \times m</math>).</p> | <p><b>Lungimea listei de adiacență</b> a nodului <math>i</math> neizolat, cu eticheta mai mică decât <math>n</math>, se calculează la fel ca și în cazul grafului neorientat. Pentru nodul <math>n</math> lungimea listei de adiacență este egală cu (<math>n+m+1 - L[1][n]</math>).</p> <p><b>Gradul extern al nodului</b> <math>i</math> este egal cu lungimea listei de adiacență a nodului.</p> <p><b>Gradul intern al nodului</b> <math>i</math> este egal, cu numărul de apariții ale etichetei nodului în a doua secțiune a primei linii a matricei (<math>L[0][j]=i</math>, cu <math>j=n+1, n+m</math>).</p> |
| <p><b>Nodurile adiacente nodului</b> <math>i</math> sunt nodurile a căror etichetă apare în lista de adiacență a nodului <math>i</math>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | <p><b>Succesorii nodului</b> <math>i</math> sunt nodurile a căror etichetă apare în lista de adiacență a nodului <math>i</math>.</p> <p><b>Predecesorii nodului</b> <math>i</math> sunt nodurile <math>j</math> în a căror listă de adiacență apare nodul <math>i</math>.</p> <p><b>Nodurile adiacente nodului</b> <math>i</math> sunt date de reuniunea dintre mulțimea succesorilor și mulțimea predecesorilor nodului.</p>                                                                                                                                                                                        |
| <p><b>Numărul de vecini ai nodului</b> <math>i</math> este egal cu gradul nodului.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | <p><b>Numărul de vecini ai nodului</b> <math>i</math> este egal cu cardinalul mulțimii de noduri adiacente nodului <math>i</math>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <p><b>Muchia</b> <math>[i,j]</math> a grafului reprezintă nodul <math>i</math> și un nod <math>j</math> din lista de adiacență a nodului <math>i</math> din prima linie a matricei (<math>L[0][j] \in L_i</math>).</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | <p><b>Arcul</b> <math>[i,j]</math> al grafului reprezintă nodul <math>i</math> și un nod <math>j</math> din lista de adiacență a nodului <math>i</math> din vectorul <math>L</math> (<math>L[0][j] \in L_i</math>).</p>                                                                                                                                                                                                                                                                                                                                                                                              |

### Temă



1. Scrieți lista de adiacență a grafului neorientat  $G_4$ . Folosind informațiile din lista de adiacență, determinați: gradul nodului 5, nodurile izolate și nodurile terminale.
2. Scrieți lista de adiacență a grafului orientat  $G_{13}$ . Ce proprietate are acest graf?
3. Scrieți lista de adiacență a grafului orientat  $G_7$ . Folosind informațiile din lista de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 4, succesorii și predecesorii nodului 2 și predecesorii nodului 3.
4. Scrieți lista de adiacență a grafului orientat  $G_{10}$ . Din lista de adiacență, determinați: gradul intern al nodului 5, gradul extern al nodului 2, nodurile adiacente nodului 5, succesorii și predecesorii nodului 4, nodurile terminale și nodurile izolate.
5. Scrieți lista de adiacență a grafului orientat  $G_{16}$  din figura 10.
6. Scrieți lista de adiacență a grafului  $G_{11}$ . Cum identificați, în lista de adiacență, nodul sursă al grafului?
7. Scrieți lista de adiacență a grafului  $G_{12}$ . Cum identificați – în lista de adiacență – nodul destinație al grafului?

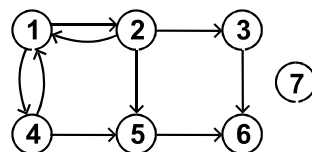


Fig 10



## Implementarea algoritmilor pentru reprezentarea grafurilor cu lista de adiacență

### 1. Crearea listei de adiacență prin citirea listelor de vecini din fișier. Determinarea gradului unui nod. Obținerea matricei de adiacență din matricea listei de adiacență.

Într-un fișier text se găsesc următoarele informații despre un graf neorientat (și varianta orientat): pe prima linie, valorile pentru numărul de noduri  $n$  și numărul de muchii  $m$ , despărțite prin spații, iar pe următoarele  $n$  linii, despărțite prin spații, numărul de vecini ai unui nod și lista vecinilor (și varianta numărul de succesori ai unui nod și lista succesorilor). Se citesc din fișierul text aceste informații și se creează matricea listei de adiacență. Se afișează nodurile cu gradul cel mai mare (și varianta cu gradul extern cel mai mare). Se obține matricea de adiacență din lista de adiacență. Se salvează apoi matricea de adiacență într-un fișier text. În fișierul text se scrie pe primul rând ordinul grafului, iar pe următoarele rânduri – liniile matricei de adiacență. Funcția `citeste()` se folosește pentru a citi listele de adiacență din fișier, funcția `grad()` pentru a determina gradul (și varianta gradul extern) al unui nod, funcția `grad max()` pentru a determina gradul maxim (și varianta gradul extern maxim) al nodurilor din graf, funcția `transpune()` pentru a obține matricea de adiacență din lista de adiacență, iar funcția `scrie()` pentru a scrie matricea de adiacență în fișier. Informațiile se citesc din fișierul text *graf13.txt*, pentru graful neorientat, și *graf14.txt*, pentru graful orientat, și se salvează în fișierul text *graf15.txt*, pentru graful neorientat, și *graf16.txt*, pentru graful orientat. Fișierele text *graf13.txt* și *graf14.txt* se creează cu un editor de texte. Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

#### Graful neorientat și graful orientat

```
#include<fstream.h>
int n,m,L[3][50],a[10][10];
//pentru graful neorientat
fstream f1("graf13.txt",ios::in),f2("graf15.txt",ios::out);
//pentru graful orientat
//fstream f1("graf14.txt",ios::in),f2("graf16.txt",ios::out);
void citeste()
{int i,j,x,k; f1>>n>>m;
 for(i=1;i<=n;i++) L[1][i]=i; j=n+1;
 for(i=1;i<=n;i++)
 {f1>>x;
 if (x==0) L[2][i]=0;
 else {L[2][i]=j;
 for (k=1;k<=x;k++) {f1>>L[1][j];
 if(k!=x) L[2][j]=j+1;
 else L[2][j]=0;
 j++;}}}

 f1.close();}
int grad(int i)
{int g,j=L[2][i];
 if (L[2][i]==0) g=0;
 else {g=1;
 while (L[2][j]!=0) {g++;j++;}}
 return g;}
int grad max()
{int i,max=0;
 for (i=1; i<=n; i++) if (grad(i)>max) max=grad(i);
 return max;}
```

```

void transpune()
{int i,j;
 for (i=1; i<=n; i++)
 {j=L[2][i];
 if (L[2][i]!=0)
 {while (L[2][j]!=0) {a[i][L[1][j]]=1; j++;}
 a[i][L[1][j]]=1;}}
void scrie()
{int i,j; f2<<n<<<endl;
 for (i=1; i<=n; i++)
 {for (j=1; j<=n; j++) f2<<a[i][j]<<" ";
 f2<<endl;}
f2.close();}
void main()
{int i; citeste();
 cout<<"Gradul cel mai mare este "<<grad max()<<endl;
 cout<<"Nodurile cu gradul cel mai mare sunt: ";
 for (i=1; i<=n; i++) if (grad(i)==grad max()) cout<<i<<" ";
 transpune(); scrie();}

```

## 2. Crearea matricei listei de adiacență din matricea de adiacență. Salvarea listelor de adiacență într-un fișier.

Se citește din fișierul creat anterior (*graf15.txt*, respectiv *graf16.txt*) matricea de adiacență a grafului, și se obține din ea matricea listei de adiacență. Se salvează lista de adiacență într-un fișier text (*graf17.txt*, respectiv *graf18.txt*), astfel: pe prima linie, valorile pentru numărul de noduri  $n$  și numărul de muchii  $m$ , despărțite prin spații, iar apoi, pe următoarele  $n$  linii, despărțite prin spații, numărul de vecini ai unui nod și lista vecinilor pentru graful neorientat, respectiv numărul de succesori ai unui nod și lista succesorilor pentru graful orientat. Funcția `citeste()` se folosește pentru a citi matricea de adiacență din fișier, funcția `transpune()` pentru a obține matricea listei de adiacență din matricea de adiacență, iar funcția `scrie()` pentru a scrie listele de adiacență în fișier. Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

### Graful neorientat și graful orientat

```

#include<fstream.h>
int n,m,L[3][50],a[10][10];
//pentru graful neorientat
fstream f1("graf15",ios::in),f2("graf17.txt",ios::out);
//pentru graful orientat
//fstream f1("graf16.txt",ios::in),f2("graf18.txt",ios::out);
void citeste()
{int i,j; f1>>n;
 for (i=1;i<=n;i++)
 {for(j=1;j<=n;j++) {f1>>a[i][j]; if(a[i][j]==1) m++;}
 m=m/2; //numai pentru graful neorientat
 f1.close();}
void transpune()
{int i,j,k,g;
 for(i=1;i<=n;i++) L[1][i]=i; k=n+1;
 for(i=1;i<=n;i++)
 {g=0;
 for (j=1;j<=n;j++)

```

```

 if (a[i][j]==1) {L[1][k]=j; L[2][k]=k+1; k++; g++;}
 if (g==0) L[2][i]=0;
 else {L[2][i]=k-g; L[2][k-1]=0;}}
int grad(int i) {//este identică cu cea de la exemplul anterior }
void scrie()
{int i,j=1,k; f2<<n<<" "<<m<<endl;
 for (i=1;i<=n;i++)
 if (L[2][i]==0) f2<<0<<endl;
 else {f2<<grad(i)<<" ";
 for (k=1;k<=grad(i);k++,j++) f2<<L[1][j]<<" ";
 f2<<endl;}}
f2.close();}
void main()
{citeste(); transpune(); scrie();}

```

### Temă



1. Scrieți un program care citește listele de adiacență ale grafului din fișierul text *graf13.txt*, pentru graful neorientat, respectiv din fișierul *graf14.txt*, pentru graful orientat, construiește matricea listei de adiacență și afișează nodurile izolate.
2. Scrieți un program care citește listele de adiacență ale grafului din fișierul text *graf13.txt*, pentru graful neorientat, respectiv din fișierul *graf14.txt*, pentru graful orientat, construiește matricea listei de adiacență și afișează muchiile (arcele) grafului.
3. Scrieți un program care citește din două fișiere text, respectiv din *g5.txt* un graf orientat, reprezentat prin lista muchiilor, și din fișierul *g6.txt* un graf orientat, reprezentat prin lista vecinilor, și care verifică dacă cele două grafuri sunt identice (**Indicație**. Se reprezintă ambele grafuri prin matricea de adiacență și se compară cele două matrice de adiacență).

## 2.6.5. Grafuri speciale

Se definesc următoarele grafuri speciale:

- **graful nul**;
- **graful complet**.

### 2.6.5.1. Graful nul

**Graful  $G=(X,U)$  se numește graf nul dacă mulțimea  $U$  este vidă ( $U=\emptyset$ ), adică graful nu are muchii.**

Reprezentarea sa în plan se face prin noduri izolate.

**Exemplu:**

Graful  $N=(X,V)$  – unde mulțimea  $X=\{1,2,3,4\}$  este mulțimea nodurilor, iar mulțimea  $V=\emptyset$  este mulțimea muchiilor – este un graf nul (graful are noduri dar nu are muchii) – figura 11.

① ②

③ ④

Fig. 11

**Observație.** Matricea de adiacență a unui graf nul este matricea zero (nu conține nici un element cu valoarea 1).

### 2.6.5.2. Graful complet

**Un graf cu  $n$  noduri este un graf complet dacă are proprietatea că, oricare ar fi două noduri ale grafului, ele sunt adiacente. El se notează cu  $K_n$ .**

**Observații.**

1. Se poate construi **un singur graf neorientat complet, cu  $n$  noduri**, deoarece între două noduri,  $x$  și  $y$ , există o singură muchie  $[x,y]$ . În figura 12 este prezentat grafurile neorientate complete  $K_5$  și  $K_6$ . Numărați câte muchii au aceste grafuri.

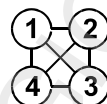


Fig. 12

**Teorema 6**

**Numărul  $m$  de muchii ale unui graf neorientat complet, cu  $n$  noduri ( $K_n$ ), este:**

$$m = n \times \frac{n-1}{2}$$

**Demonstrație.** Numărul de muchii este dat de numărul de submulțimi de 2 elemente care se pot forma dintr-o mulțime cu  $n$  elemente, adică  $m = C_n^2$ .

2. Se pot construi **mai multe grafuri orientate complete, cu  $n$  noduri**, deoarece două noduri  $x$  și  $y$  pot fi adiacente în trei situații: există arcul  $[x,y]$ , există arcul  $[y,x]$  sau există arcele  $[x,y]$  și  $[y,x]$ .

**Teorema 7**

**Numărul de grafuri orientate complete care se pot construi cu  $n$  noduri este egal cu**

$$n_k = 3^{C_n^2}$$

**Demonstrație.** Deoarece numărul de submulțimi de 2 noduri care se pot forma dintr-o mulțime de  $n$  noduri este  $a = C_n^2$ , și pentru fiecare pereche astfel definită se pot defini trei situații de adiacență, rezultă că numărul de grafuri complete care se pot construi este de  $3^a$ .

**Exemplu.** Pentru  $n=4$  se pot defini  $3^6 = 729$  grafuri orientate complete. În figura 13 sunt prezentate patru dintre acestea. Definiți alte patru grafuri complete cu 4 noduri.

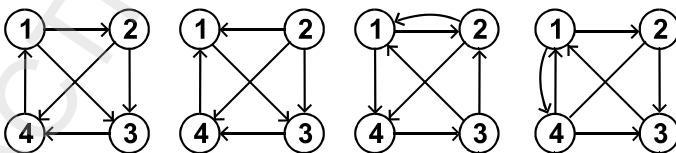


Fig. 13

**Observații**

1. În cazul matricei de adiacență a unui graf neorientat complet, valoarea fiecărui element care nu se găsește pe diagonala principală este 1.
2. În cazul matricei de adiacență a unui graf orientat complet – pentru orice pereche de noduri,  $i$  și  $j$ , diferite între ele ( $i \neq j$ ) –  $a[i][j]=1$  sau  $a[j][i]=1$ .
3. **Numărul minim de arce** într-un graf orientat, cu  $n$  noduri, este egal cu numărul de muchii ale grafului neorientat complet  $K_n$ .
4. **Numărul maxim de arce** într-un graf orientat, cu  $n$  noduri, este egal cu dublul numărului de muchii ale grafului neorientat complet  $K_n$ .

**Algoritmi pentru prelucrarea grafurilor complete**

1. **Algoritm pentru a determina numărul minim de arce care trebuie adăugate la un graf orientat, pentru a obține un graf orientat complet.**

**Algoritmul.** Se numără perechile de noduri  $i$  și  $j$  ( $i \neq j$ ) între care nu există nici un arc.

**Implementarea algoritmului** În program, informațiile despre graful orientat se citesc din fișierul text *gc.txt*: de pe prima linie numărul de noduri, și apoi, de pe următoarele rânduri matricea de adiacență.

```
#include<iostream.h>
int n,a[10][10];
fstream f("gc.txt",ios::in);
void citeste()
{//funcția prin care se citește matricea de adiacență din fișier}
void main()
{int i,j,m=0; citeste();
 for(i=2;i<=n;i++)
 for(j=1;j<i;j++) if(a[i][j]==0 && a[j][i]==0) m++;
 cout<<"Numarul de arce care trebuie adaugate este "<<m; }
```

## 2. Algoritm pentru a determina numărul maxim de noduri izolate pe care poate să le conțină un graf neorientat care are $n$ noduri și $m$ muchii.

**Algoritmul.** Se identifică **graful complet** care se poate forma astfel încât să consume cât mai multe muchii ( $m_{\max}$ ) din cele  $m$  muchii ale grafului ( $m_{\max} \leq m$ ). Graful complet astfel obținut are  $n_1$  noduri:

$$m_{\max} = n_1 \times \frac{n_1 - 1}{2} \leq m$$

Numărul de noduri  $n_1$  este partea întreagă din rădăcina pozitivă a ecuației de gradul II:

$$n_1 = \left\lfloor \frac{1 + \sqrt{1 + 8 \times m}}{2} \right\rfloor$$

Pentru diferența de muchii rămase ( $m - m_{\max}$ ) mai este necesar un nod, pentru a lega aceste muchii de nodurile grafului complet care s-a format. Numărul de noduri izolate este:  $n - n_1 - 1$ .

```
#include<iostream.h>
#include<math.h>
void main()
{int m,n,n1; cout<<"muchii= "; cin>>m; cout<<"noduri= "; cin>>n;
 n1=(int) ((1+sqrt(1+8*m))/2);
 cout<<"Numarul maxim de noduri izolate este "<<n-n1-1; }
```

**Temă**



1. Scrieți un program care citește, din fișierul text *graf\_c1.txt*, informații despre un graf neorientat (de pe prima linie, numărul de noduri, apoi matricea de adiacență), și care verifică dacă este un graf complet.
2. Scrieți un program care citește, din fișierul *graf\_c2.txt*, informații despre un graf orientat (de pe prima linie, numărul de noduri, apoi matricea de adiacență) – și care verifică dacă este un graf complet.

## 2.6.6. Grafuri derivate dintr-un graf

Se definesc următoarele grafuri:

- **graful parțial;**
- **subgraful.**

### 2.6.6.1. Graful parțial

Fie graful  $G = (X, U)$  și mulțimea  $V \subseteq U$ . Graful  $G_p = (X, V)$  se numește **graf parțial** al grafului  $G$ .

Altfel spus, un **graf parțial** al grafului  $G$  este el însuși sau un graf care s-a obținut prin eliminarea unor muchii (arce) din graful  $G$ .

**Exemple:**

1. Pentru graful neorientat  $G_1 = (X_1, U_1)$ , definit anterior, graful  $G_{1p} = (X_1, U_{1p})$ , definit astfel:

→ **mulțimea nodurilor** este  $X_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$ .

→ **mulțimea muchiilor** este  $U_{1p} = \{[1, 2], [1, 3], [1, 4], [2, 3], [2, 5], [3, 4], [6, 7]\}$ .

este subgraf al grafului  $G_1$  – figura 14.

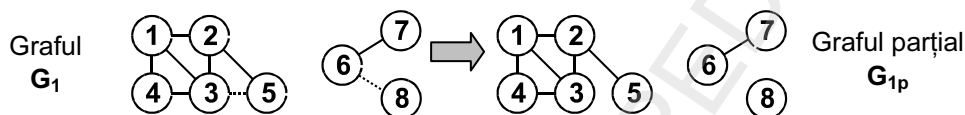


Fig. 14

2. Pentru graful orientat  $G_9 = (X_9, U_9)$  definit anterior, graful  $G_{9p} = (X_9, U_{9p})$  definit astfel:

→ **mulțimea nodurilor** este  $X_{9p} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ .

→ **mulțimea muchiilor** este  $U_{9p} = \{[2, 3], [2, 5], [2, 6], [2, 7], [4, 1], [7, 2], [8, 9], [9, 8]\}$ .

este subgraf al grafului  $G_9$  – figura 15.

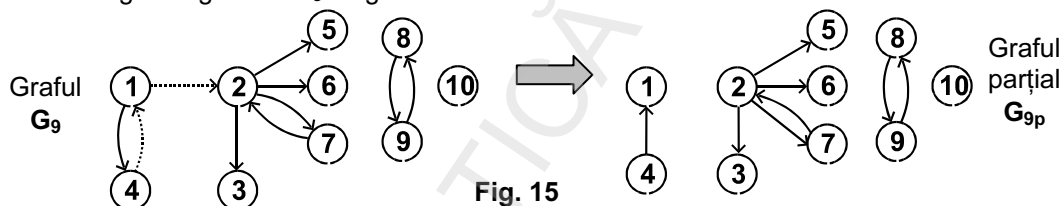


Fig. 15

### Teorema 8

**Numărul de grafuri parțiale ale unui graf cu  $m$  muchii (arce) este egal cu  $2^m$ .**

**Demonstrație.** Grafurile parțiale se pot obține: fără eliminarea unei muchii (arc) – obținându-se graful inițial; prin eliminarea unei muchii (unui arc) – obținându-se grafurile parțiale cu  $m-1$  muchii; ...; prin eliminarea a  $m-1$  muchii (arce) – obținându-se grafurile parțiale cu o muchie (un arc); și a tuturor muchiilor (arcelor) – obținându-se un graf parțial numai cu noduri și fără muchii (arce), adică graful vid:

Numărul de grafuri parțiale care se pot obține cu  $m$  muchii (arce):

$$C_m^m$$

Numărul de grafuri parțiale care se pot obține cu  $m-1$  muchii (arce):

$$C_m^{m-1}$$

Numărul de grafuri parțiale care se pot obține cu  $m-2$  muchii (arce):

$$C_m^{m-2}$$

Numărul de grafuri parțiale care se pot obține cu o muchie (un arc):

$$C_m^1$$

Numărul de grafuri parțiale care se pot obține fără nici o muchie (nici un arc):

$$C_m^0$$

Numărul total de grafuri parțiale este:  $C_m^m + C_m^{m-1} + C_m^{m-2} + \dots + C_m^1 + C_m^0 = 2^m$ .

## Algoritmi pentru prelucrarea grafurilor parțiale

### 1. Verificarea dacă un graf $G_p$ este graf parțial al unui graf $G$ .

**Algoritmul.** Se verifică dacă cele două grafuri au același număr de noduri și dacă graful  $G_p$  nu conține muchii care nu există în graful  $G$ .

**Implementarea algoritmului.** Se citesc, din două fișiere text, *g1p.txt* și *g2p.txt*, informații despre două grafuri neorientate (sau orientate): de pe prima linie, numărul de noduri, și apoi, de pe următoarele rânduri, matricea de adiacență. Matricele de adiacență ale celor două grafuri sunt  $a1$  și  $a2$ , cu dimensiunea  $n$ , respectiv  $m$ . Funcția `grafp()` verifică dacă graful  $G_p$  este graf parțial al grafului  $G$ .

```
#include<iostream.h>
fstream f1("g1p.txt",ios::in),f2("g2p.txt",ios::in);
int m,n,a1[10][10],a2[10][10];
int grafp()
{int i,j;
 if (m!=n) return 0;
 else
 for (i=1;i<=n;i++)
 for(j=1;j<=n;j++) if (a2[i][j]==1 && a1[i][j]==0) return 0;
 return 1;}
void main()
{int i,j; f1>>n;
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++) f1>>a1[i][j] f1.close();
 f2>>m;
 for(i=1;i<=m;i++)
 for(j=1;j<=m;j++) f2>>a2[i][j]; f2.close();
 if(grafp()) cout<<"este graf partial ";
 else cout<<"nu este graf partial";}
```

### 2. Obținerea unui graf parțial $G_p$ al unui graf $G$ .

**Algoritmul.** Se elimină din graful  $G$  muchiile, în funcție de condiția impusă de problemă. Reprezentarea cea mai adecvată pentru graf este matricea de adiacență, în care se atribuie valoarea 0 elementelor  $a[i][j]$  și  $a[j][i]$ , corespunzătoare muchiilor  $[i,j]$  care trebuie eliminate..

**Implementarea algoritmului.** Se citesc, din fișierul text *g3p.txt*, informații despre graful neorientat: de pe prima linie numărul de noduri  $n$  și eticheta unui nod  $x$ , și apoi, de pe următoarele rânduri, matricea de adiacență a grafului. Informațiile despre graful parțial obținut se scriu în fișierul text *g4p.txt*, astfel: pe primul rând, numărul de noduri  $n$  și numărul de muchii  $m$ , și, pe următoarele  $m$  rânduri, muchiile, sub formă de perechi de etichete de noduri, despărțite prin spațiu. Cerința este să se obțină subgraful, prin eliminarea tuturor muchiilor care au la extremități un nod cu grad par și nodul  $x$ . În vectorul  $v$  se memorează nodurile care au grad par. Funcția `citește()` se folosește pentru a citi informațiile despre matricea de adiacență a grafului din fișierul text, funcția `scrie()` pentru a scrie în fișierul text informațiile despre lista muchiilor grafului parțial, funcția `grad()` pentru a determina gradul unui nod, iar funcția `graf_partial()` pentru a obține graful parțial. Pentru a obține graful parțial, se caută toate muchiile care au la extremități un nod cu gradul par și nodul  $x$  (muchii  $[v[i],x]$  pentru care  $a[v[i]][x]==1$ ) și se elimină aceste muchii. Pentru testarea programului se va folosi graful  $G_3$  și nodul 6.



```
#include<iostream.h>
fstream f1("g3p.txt",ios::in),f2("g4p.txt",ios::out);
int a[20][20],n,m,x,v[10];
void citeste()
{//funcția prin care se citește matricea de adiacență din fișier}
int grad(int i)
{int j,g=0;
 for (j=1;j<=n;j++) g+=a[i][j]; return g;}
void graf_partial()
{int i,k=0;
 for (i=1;i<=n;i++)
 if (grad(i)%2==0) {k++; v[k]=i;}
 for (i=1;i<=k;i++)
 if (a[v[i]][x]==1) {a[v[i]][x]=0; a[x][v[i]]=0; m--;}
void scrie()
{int i,j; f2<<n<<" "<<m<<endl;
 for (i=1;i<=n;i++)
 for (j=1;j<i;j++) if (a[i][j]==1) f2<<i<<" "<<j<<endl;
 f2.close();}
void main()
{citeste(); graf_partial(); scrie();}
```

### Temă



1. Scrieți un program care citește, din fișierul text *graf16.txt*, matricea de adiacență a grafului orientat  $G_8$  și care generează toate **grafurile parțiale** care se pot obține. Informațiile despre grafurile parțiale se vor salva în fișierul text *graf\_p16.txt*. Pentru fiecare graf parțial generat se va scrie, pe un rând, textul *Graful parțial*, urmat de numărul de ordine al grafului generat, și, pe rândul următor, textul *Arcele* urmat de lista de arce a grafului generat.
2. Scrieți un program care citește, din fișierul text *graf19.txt*, matricea de adiacență a unui graf neorientat, și care generează un graf parțial, prin eliminarea muchiilor care au la extremități un nod ce are gradul minim și un nod ce are gradul maxim în graf. Informațiile despre graful parțial obținut se scriu într-un fișier text, sub forma listei de muchii. Pentru testarea programului se va folosi graful  $G_4$ .

### 2.6.6.2. Subgraful

Fie graful  $G = (X, U)$ . Graful  $G_s = (Y, V)$  se numește **subgraf** al grafului  $G$  dacă  $Y \subseteq X$  (subgraful conține numai noduri ale grafului) și muchiile (arcele) din mulțimea  $V$  sunt toate muchiile (arcele) din mulțimea  $U$  care au ambele extremități în mulțimea de noduri  $Y$ . Se spune că subgraful  $G_s$  este **indus** sau **generat** de mulțimea de noduri  $Y$ .

Altfel spus, un subgraf al grafului  $G$  este el însuși sau un graf care s-a obținut prin suprimarea din graful  $G$  a unor noduri și a tuturor muchiilor (arcelor) incidente cu aceste noduri.

#### Exemplu:



1. Pentru graful neorientat  $G_1 = (X_1, U_1)$ , definit anterior, graful  $G_{1s} = (Y_1, V_1)$ , definit astfel:
  - **mulțimea nodurilor** este  $Y_1 = \{1, 2, 4, 5, 6, 7\}$ .
  - **mulțimea muchiilor** este  $V_1 = \{[1, 2], [1, 4], [2, 5], [6, 7]\}$ .
 este subgraf al grafului  $G_1$  – figura 16.

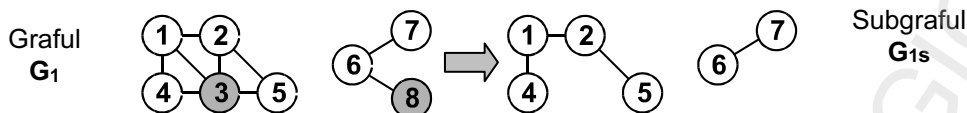


Fig. 16

2. Pentru graful orientat  $G_{10}=(X_9, U_9)$ , definit anterior, graful  $G_{9s}=(Y_9, V_9)$  definit astfel:

→ mulțimea nodurilor este  $Y_9=\{1,2,3,6,7,9,10\}$ .

→ mulțimea muchiilor este  $V_9=\{[1,2], [2,1], [2,3], [2,6], [2,7], [7,2]\}$ .

este subgraf al grafului  $G_9$  – figura 17.

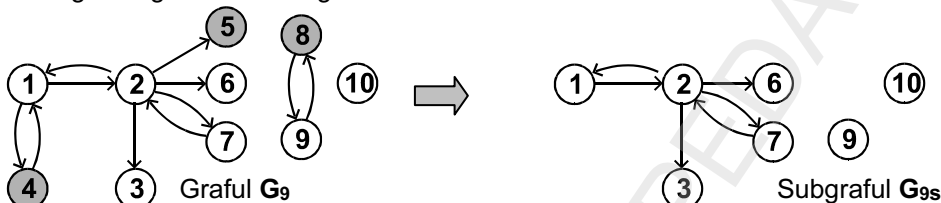


Fig. 17

### Teorema 9

**Numărul de subgrafuri ale unui graf cu  $n$  noduri este egal cu  $2^n - 1$ .**

**Demonstrație.** Subgrafurile se pot obține: prin eliminarea niciunui nod (obținându-se graful inițial); a unui nod (obținându-se subgrafurile cu  $n-1$  noduri); ...; a  $n-1$  noduri (obținându-se subgrafurile cu un nod):

Numărul de subgrafuri care se pot obține cu  $n$  noduri:

$$C_n^n$$

Numărul de subgrafuri care se pot obține cu  $n-1$  noduri:

$$C_n^{n-1}$$

Numărul de subgrafuri care se pot obține cu  $n-2$  noduri:

$$C_n^{n-2}$$

.....  
Numărul de subgrafuri care se pot obține cu 1 nod:

$$C_n^1$$

Numărul total de subgrafuri este:  $C_n^n + C_n^{n-1} + C_n^{n-2} + \dots + C_n^1 = 2^n - 1$ .

### Algoritmi pentru prelucrarea subgrafurilor

#### 1. Generarea tuturor subgrafurilor unui graf.

**Algoritmul.** Se folosește **metoda backtracking**. În stivă se generează nodurile subgrafului. Deoarece subgraful poate avea  $p$  noduri ( $1 \leq p \leq n$ ), se va apela repetat subprogramul **bt()**, la fiecare apel generându-se variantele cu  $p$  noduri. Fiecare apel al subprogramului **bt()** trebuie să genereze  $C_n^p$  de noduri din graf. Condiția ca un nod adăugat în stivă să facă parte din soluție este ca el să nu mai existe în stivă. La fiecare apel al subprogramului, soluția se obține atunci când în stivă s-au generat cele  $p$  noduri ale subgrafului. Pentru nodurile generate în stivă se afișează muchiile (arcele) care există în graf.

**Implementarea algoritmului.** Se citesc, din fișierul text *graf1.txt*, matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt*, matricea de adiacență a unui graf orientat. Pentru fiecare subgraf generat, se afișează numărul de noduri și muchiile (arcele). În variabila **nr** se contorizează numărul de subgrafuri generate. În funcția

**tipar()** se afișează subgraful generat. Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

```
#include<fstream.h>
fstream f("graf1.txt",ios::in);
// fstream f("graf2.txt",ios::in); pentru graful orientat
typedef int stiva[100];
int n,p,k,ev,as,a[10][10],nr;
stiva st;
void citeste()
{///funcția prin care se citește matricea de adiacență din fișier}
void init() {st[k]=0;}
int succesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;}
 else return 0;}

int valid()
{if (k>1 && st[k]<st[k-1]) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
 return 1;}
int solutie() {return k==p;}
void tipar()
{int i,j; nr++;
 cout<<"Subgraful "<<nr<<endl<<"Nodurile:";
 for (i=1;i<=p;i++) cout<<st[i]<<" "; cout<<endl;
 cout<<"Muchiile: "; // cout<<"Arcele: "; pentru graful orientat
 for(i=1;i<=p;i++)
 for(j=i+1;j<=p;j++) // for(j=1;j<=p;j++) pentru graful orientat
 if (a[st[i]][st[j]]==1) cout<<st[i]<<"-"<<st[j]<<" ";
 cout<<endl;}
void bt()
{///partea fixă a algoritmului backtracking}
void main()
{citeste();
 for(p=n;p>=1;p--) bt();}
```

## 2. Verificarea dacă un graf $G_s$ este subgraf al unui graf $G$ .

**Algoritmul.** Se verifică dacă:

- numărul de noduri din graful  $G_s$  este mai mic sau cel mult egal cu numărul de noduri din graful  $G$ ;
- etichetele nodurilor din graful  $G_s$  există printre etichetele grafului  $G$ ;
- între nodurile din graful  $G_s$  există muchiile dintre nodurile din graful  $G$  și numai acelea.

**Implementarea algoritmului.** Se citesc, din două fișiere text *g1s.txt* și *g2s.txt*, informații despre două grafuri neorientate (orientate): de pe prima linie, numărul de noduri, și apoi, de pe următoarele rânduri, matricea de adiacență. În fișierul *g2s.txt*, pe ultimul rând, după matricea de adiacență, este memorat un șir de numere care reprezintă etichetele nodurilor din acest graf. Matricele de adiacență ale celor două grafuri sunt  $a_1$  și  $a_2$ , cu dimensiunea  $n$ , respectiv  $m$ . În vectorul  $v$  se memorează etichetele nodurilor celui de al doilea graf. Funcția **subgraf()** verifică dacă graful  $G_s$  este subgraf al grafului  $G$ .

```
#include<iostream.h>
int m,n,a1[10][10],a2[10][10],v[10];
fstream f1("g1s.txt",ios::in),f2("g2s.txt",ios::in);
```

```

int subgraf()
{int i,j;
 if (m>n) return 0;
 else
 {for (i=1;i<=m;i++) if (v[i]>n) return 0;
 for(i=1;i<=m;i++)
 for(j=1;j<=m;j++) if (a2[i][j]!=a1[v[i]][v[j]]) return 0;}
 return 1;}
void main()
{int i,j; f1>>n;
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++) f1>>a1[i][j];
 f1.close(); f2>>m;
 for(i=1;i<=m;i++)
 for(j=1;j<=m;j++) f2>>a2[i][j];
 for(i=1;i<=m;i++) f2>>v[i];
 f2.close();
 if(subgraf()) cout<<"este subgraf ";
 else cout<<"nu este subgraf";}

```

### Temă



1. Scrieți un program care citește, din fișierul text *graf16.txt*, matricea de adiacență a unui graf orientat – și care generează toate subgrafurile care se pot obține. Informațiile despre subgrafuri se vor salva în fișierul text *graf\_s16.txt*. Pentru fiecare subgraf generat se va scrie, pe un rând, textul *Subgraful x are p noduri* (unde *x* este numărul de ordine al subgrafului generat, iar *p* numărul de noduri) și, pe rândul următor, textul *Arcele* – urmat de lista de arce a subgrafului generat.
2. Scrieți un program care citește, din fișierul text *graf16.txt*, matricea de adiacență a unui graf orientat – și care generează subgraful care se obține prin eliminarea nodului care are cei mai mulți vecini.

## 2.6.7. Conexitatea grafurilor

### 2.6.7.1. Lanțul

Într-un graf  $G = (X, U)$  se definește **lanțul** ca fiind o succesiune de noduri care au proprietatea că, oricare ar fi două noduri succesive, ele sunt adiacente.

#### Graful neorientat

Dacă mulțimea nodurilor unui graf neorientat este  $X = \{x_1, x_2, \dots, x_n\}$ , un lanț de la nodul  $l_1$  la nodul  $l_k - L(l_1, l_k)$  – va fi definit prin mulțimea  $L(l_1, l_k) = \{l_1, l_2, \dots, l_i, \dots, l_k\}$ , unde  $l_i \in X$  pentru orice  $i$  ( $1 \leq i \leq k$ ), iar muchiile  $[l_1, l_2], [l_2, l_3], [l_3, l_4], \dots, [l_{k-1}, l_k] \in U$ . Lanțul poate fi interpretat ca un traseu prin care se parcurg anumite muchii ale grafului, traseul fiind ordinea în care se parcurg aceste muchii:  $[l_1, l_2], [l_2, l_3], [l_3, l_4], \dots, [l_{k-1}, l_k]$ . Fiecare pereche de noduri succesive din lanț reprezintă parcurgerea unei muchii. Dacă există  $L(x_i, x_j)$ , se spune că **nodul  $x_j$  este accesibil din nodul  $x_i$** .

#### Graful orientat

Dacă mulțimea nodurilor unui graf orientat este  $X = \{x_1, x_2, \dots, x_n\}$ , un lanț de la nodul  $l_1$  la nodul  $l_k - L(l_1, l_k)$  – va fi definit prin mulțimea  $L(l_1, l_k) = [l_1, l_2, \dots, l_i, \dots, l_k]$ , unde  $l_i \in X$ , pentru orice  $i$  ( $1 \leq i \leq k$ ). Arcele  $[l_1, l_2], [l_2, l_3], [l_3, l_4], \dots, [l_{k-1}, l_k]$  au proprietatea că, oricare ar fi două arce

successive, ele au o extremitate comună. **La definirea unui lanț, nu se ține cont de orientarea arcelor, ci numai de faptul că două noduri sunt legate de un arc.**

### Terminologie:

- **Lungimea unui lanț** reprezintă numărul de parcurgeri ale muchiilor, respectiv arcelor. De exemplu, lungimea lanțului  $L(l_1, l_k)$  este  $k-1$ . Dacă o muchie (un arc) este parcursă de mai multe ori, se va număra fiecare dintre parcurgerile sale.
- **Lanț de lungime minimă** dintre nodul  $x_i$  și nodul  $x_j$  –  $L_{\min}(x_i, x_j)$  – este lanțul cu numărul minim de muchii (arce), din mulțimea nevidă de lanțuri  $L(x_i, x_j)$ .
- **Extremitățile unui lanț** sunt formate din nodul cu care începe și nodul cu care se termină lanțul ( $l_1$  și  $l_k$ ).
- **Sublanțul** este format dintr-un șir continuu de noduri din lanț. De exemplu, pentru lanțul  $L(l_1, l_k) = \{l_1, l_2, \dots, l_i, \dots, l_j, \dots, l_k\}$ ,  $L_s(l_i, l_j) = \{l_i, l_{i+1}, \dots, l_{j-1}, l_j\}$  este un sublanț al lanțului  $L$ .

### Exemple:

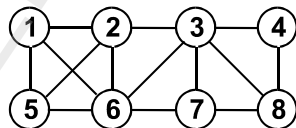


Fig. 18

G<sub>17</sub>

1. Pentru graful neorientat  $G_{17} = (X_{17}, U_{17})$ , definit astfel:

→ **mulțimea nodurilor** este  $X_{17} = \{1, 2, 3, 4, 5, 6, 7, 8\}$ .

→ **mulțimea muchiilor** este  $U_{17} = \{[1, 2], [1, 5], [1, 6], [2, 3], [2, 5], [2, 6], [3, 4], [3, 6], [3, 7], [3, 8], [4, 8], [5, 6], [6, 7], [7, 8]\}$ .

$L_1(1, 7) = \{1, 2, 3, 8, 4, 3, 6, 5, 2, 3, 7\}$  este un lanț între nodul cu eticheta 1 și nodul cu eticheta 7 (figura 18). Lungimea lanțului este 10.

G<sub>18</sub>

2. Pentru graful orientat  $G_{18} = (X_{18}, U_{18})$ , definit astfel:

→ **mulțimea nodurilor** este  $X_{18} = \{1, 2, 3, 4, 5, 6, 7\}$ .

→ **mulțimea arcelor** este  $U_{18} = \{[1, 2], [1, 4], [2, 3], [2, 4], [3, 6], [3, 7], [4, 1], [4, 5], [5, 2], [5, 4], [6, 3], [6, 5], [7, 6]\}$ .

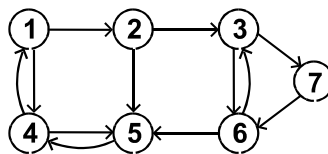


Fig. 19

$L_1(1, 5) = \{1, 2, 5, 6, 3, 6, 7, 6, 5\}$  este un lanț între nodul cu eticheta 1 și nodul cu eticheta 5 (figura 19). Lungimea lanțului este 8.

### Clasificarea lanțurilor – în funcție de:

#### Noduri

##### Elementare

Conțin numai **noduri distincte două câte două**:  $l_1 \neq l_2; l_1 \neq l_3; \dots; l_1 \neq l_k; l_2 \neq l_3; \dots; l_{k-1} \neq l_k$  – același nod poate fi parcurs o singură dată.

##### Neelementare

Conțin **noduri care se repetă** – prin același nod se poate trece de mai multe ori.

#### Muchii (arce)

##### Simple

Toate muchiile (arce) din lanț **sunt diferite între ele**:  $[l_1, l_2] \neq [l_2, l_3]; [l_1, l_2] \neq [l_3, l_4]; \dots; [l_1, l_2] \neq [l_{k-1}, l_k]; \dots; [l_{k-2}, l_{k-1}] \neq [l_{k-1}, l_k]$  – aceeași muchie (arc) poate fi parcursă o singură dată.

##### Compuse

În lanț se pot **repetă unele muchii** (arce) – aceeași muchie (arc) poate fi parcursă de mai multe ori.

**Exemple:****1. Pentru graful neorientat  $G_{17}$ :**

- Lanțul  $L_1(1,7)$  definit anterior este un **lanț neelementar**, deoarece se repetă nodul cu eticheta 3. Este un **lanț compus**, deoarece în lanț se repetă muchia  $[2,3]$ .
- Lanțul  $L_2(1,7) = \{1, 2, 3, 6, 7\}$  este un **lanț elementar** deoarece fiecare nod este parcurs o singură dată.
- Lanțul  $L_3(1,7) = \{1, 6, 3, 2, 6, 7\}$  este un **lanț simplu**, deoarece nici o muchie nu se repetă, dar este un **lanț neelementar**, deoarece se repetă nodul cu eticheta 6.

**2. Pentru graful orientat  $G_{18}$ :**

- Lanțul  $L_1(1,5)$  definit anterior este un **lanț neelementar**, deoarece se repetă nodul cu eticheta 6. Este un **lanț compus**, deoarece în lanț se repetă arcul  $[6,7]$ .
- Lanțul  $L_2(1,5) = \{1, 2, 3, 7, 6, 5\}$  este un **lanț elementar**, deoarece fiecare nod este parcurs o singură dată.
- Lanțul  $L_3(1,5) = \{1, 2, 4, 5, 2, 3, 6, 5\}$  este un **lanț simplu**, deoarece nici un arc nu se repetă, dar este un **lanț neelementar** deoarece se repetă nodul cu eticheta 2.

**Teorema 10**

**Dacă un graf conține un lanț între două noduri,  $x$  și  $y$ , atunci conține un lanț elementar între nodurile  $x$  și  $y$ .**

**Demonstrație.** Considerăm lanțul  $L(x,y)=\{x, l_1, l_2, \dots, l_k, y\}$ . Dacă lanțul nu este elementar, el conține cel puțin un nod  $z$  care se repetă – există  $i$  și  $j$ , astfel încât  $l_i = l_j = z$ :  $L(x,y)=\{x, l_1, l_2, \dots, l_i, \dots, l_j, \dots, l_k, y\}$ . Nodul  $z$  aparținând lanțului  $L(x,y)$ , înseamnă că el este accesibil din nodul  $x$ , iar nodul  $y$  este accesibil din nodul  $z$ . Înseamnă că din lanț se poate elimina sublanțul care leagă nodul  $z$  de el însuși:  $L(x,y)=\{x, l_1, l_2, \dots, l_i, l_{j+1}, \dots, l_k, y\}$ . În același mod, se pot elimina din lanț, toate sublanțurile care leagă un nod de el însuși, obținându-se în final un lanț în care fiecare nod nu apare decât o singură dată, adică un lanț elementar.

**Exemplu:**

În graful neorientat  $G_{17}$ , lanțul  $L_1(1,7)$  este un lanț neelementar. Prin eliminarea din acest lanț a sublanțului  $\{8, 4, 3\}$ , se obține lanțul  $\{1, 2, 3, 6, 5, 2, 3, 7\}$ , care este un lanț neelementar. Prin eliminarea din acest lanț a sublanțului  $\{6, 5, 2, 3\}$ , se obține lanțul  $\{1, 2, 3, 7\}$ , care este un lanț elementar.

**Teorema 11**

**Dacă un lanț este elementar, atunci este și lanț simplu.**

**Demonstrație** – prin reducere la absurd. Presupunem că lanțul elementar este și lanț compus. Dacă este lanț compus, el trebuie să parcurgă de două ori aceeași muchie (arc), ceea ce ar însemna să treacă de două ori prin nodurile adiacente muchiei (arcului). Lanțul fiind elementar, nu trece însă de două ori prin același nod.

**Algoritmi pentru determinarea lanțurilor elementare într-un graf****1. Afișarea lanțurilor elementare dintre două noduri.**

**Algoritmul.** Pentru generarea lanțurilor elementare între nodul  $x$  și nodul  $y$ , se folosește metoda **backtracking**. Nodurile lanțului elementar vor fi generate în stivă. Primul nod din stivă este nodul  $x$ . Condiția ca un nod adăugat în stivă să facă parte din soluție (să fie valid) este să formeze o muchie (un arc) cu nodul adăugat anterior în stivă, și să nu mai existe în stivă (condiția ca lanțul să fie elementar). Soluția se obține atunci când nodul adăugat în stivă este nodul  $y$ .

**Implementarea algoritmului.** Se citesc, din fișierul text *graf1.txt*, matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt*, matricea de adiacență a unui graf orientat. Se mai citesc de la tastatură două valori numerice care reprezintă etichetele a două noduri din graf. Se caută toate lanțurile elementare care există între cele două noduri și se afișează. Dacă nu există nici un lanț elementar, se afișează un mesaj. Funcția **citeste()** se folosește pentru a citi matricea de adiacență din fișier. Variabila **este** se folosește pentru a verifica dacă s-a găsit un lanț elementar între cele două noduri (are valoarea 0 – *False*, dacă nu s-a găsit un lanț elementar). Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

```
#include<fstream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,y,este=0;
stiva st;
fstream f("graf1.txt",ios::in);
void citeste()
{//funcția prin care se citește matricea de adiacență din fișier}
void init() {st[k]=0;}
int sucesor()
{if (st[k]<n) {st[k]=st[k]+1; return 1;}
 else return 0;}

int valid()
{int i;
 if(k>1) //condiția ca două noduri succesive să fie adiacente
 if (a[st[k-1]][st[k]]==0) return 0;
 //if (a[st[k]][st[k-1]]==0 && a[st[k-1]][st[k]]==0) return 0;
 //pentru graful orientat
 for (i=1;i<k;i++) //condiția ca lanțul să fie elementar
 if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return st[k]==y;}
//se obține soluția atunci când ultimul nod adăugat în stivă este y
void tipar ()
{int i; este=1;
 for (i=1;i<=k;i++) cout<<st[i]<<" ";
 cout<<endl;}
void bt() //partea fixa a algoritmului
{ k=2; init();
 while (k>1)
 {as=1; ev=0;
 while(as && !ev)
 {as=sucesor();
 if(as) ev=valid();}
 if(as)
 if (solutie()) tipar();
 else {k++; init();}
 else k--;}}
void main()
{citeste();
 cout<<"x= "; cin>>x; cout<<"y= "; cin>>y; st[1]=x; bt();
 if (!este) cout<<"Nu exista lant";}
```



## 2. Afișarea tuturor lanțurilor elementare din graf.

**Algoritmul.** Pentru generarea tuturor lanțurilor elementare din graf, se vor genera lanțurile elementare dintre nodul  $x$  ( $1 \leq x \leq n$ ) și nodul  $y$  ( $1 \leq y \leq n$ ), folosind metoda **backtracking**, care se va apela pentru fiecare pereche de noduri  $x$  și  $y$  ( $x \neq y$ ).

**Implementarea algoritmului.** Se citesc, din fișierul text *graf1.txt*, matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt*, matricea de adiacență a unui graf orientat. Se caută toate lanțurile elementare care există în graf și se afișează. Dacă nu există nici un lanț elementar, se afișează un mesaj. Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

```
#include<fstream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,y,este=0;
stiva st;
fstream f("graf1.txt",ios::in);
void citeste() { //la fel ca în exemplul precedent }
void init() { //la fel ca în exemplul precedent }
int succesor() { //la fel ca în exemplul precedent }
int valid() { //la fel ca în exemplul precedent }
int solutie() { //la fel ca în exemplul precedent }
void tipar () { //la fel ca în exemplul precedent }
void bt()
{//partea fixa a algoritmului backtracking - la fel ca în exemplul
//precedent}
void main()
{citeste();
for (x=1;x<=n;x++)
for (y=1;y<=n;y++) if (x!=y) {st[1]=x; bt();}
if (!este) cout<<"Nu exista lanturi elementare";}
```

### Temă



Scrieți câte un program care citește, dintr-un fișier text, lista muchiilor unui graf neorientat (*graf3.txt*) sau, variantă graf orientat (*graf4.txt*), și care:

- Verifică dacă un șir de  $k$  numere citite de la tastatură reprezintă un lanț elementar pentru graf.
- Caută toate lanțurile elementare cu lungimea  $d$  și le afișează. Valoarea numerică  $d$  se citește de la tastatură. Dacă nu există nici un lanț elementar cu lungimea  $d$ , se afișează un mesaj.
- Caută toate lanțurile elementare, cu lungimea cea mai mică, ce există între două noduri,  $x$  și  $y$ , și le afișează. Etichetele nodurilor se citesc de la tastatură. Dacă nu există nici un lanț elementar, se afișează un mesaj..
- Caută toate lanțurile elementare, între două noduri  $x$  și  $y$ , care trec printr-un nod  $z$  care are gradul minim în graf. Etichetele celor două noduri se citesc de la tastatură. Dacă nu există nici un lanț elementar, se afișează un mesaj.

### 2.6.7.2. Ciclul

Un lanț care are toate muchiile distincte două câte două și extremități care coincid – se numește **ciclu**.

Ciclu este un lanț simplu ( $[l_1, l_2] \neq [l_2, l_3]; [l_1, l_2] \neq [l_3, l_4]; \dots; [l_1, l_2] \neq [l_{k-1}, l_k]; \dots; [l_{k-2}, l_{k-1}] \neq [l_{k-1}, l_k]$ ), în care extremitățile coincid:  $l_1 = l_k$ .

Un graf fără cicluri se numește **graf aciclic**.

Dacă toate nodurile unui ciclu sunt distincte două câte două, cu excepția extremităților, ciclul se numește **ciclu elementar**.

### Exemple:

#### 1. Pentru graful neorientat $G_{17}$ :

- Lanțul  $L_4(1,1) = \{1, 2, 3, 6, 2, 1\}$  nu este un ciclu deoarece în lanț se repetă muchia  $[1,2]$ .
- Lanțul  $L_5(1,1) = \{1, 2, 6, 3, 7, 6, 1\} = C_1$  este un ciclu neelementar deoarece se repetă nodul cu eticheta 6.
- Lanțul  $L_6(1,1) = \{1, 2, 3, 7, 6, 1\} = C_2$  este un ciclu elementar deoarece nu se repetă nici un nod.

#### 2. Pentru graful orientat $G_{19}$ :

- Lanțul  $L_4(1,1) = \{1, 2, 4, 5, 2, 1\}$  nu este un ciclu deoarece în lanț se repetă arcul  $[1,2]$ .
- Lanțul  $L_5(1,1) = \{1, 2, 5, 6, 3, 2, 1\} = C_1$  este un ciclu neelementar deoarece se repetă nodul cu eticheta 2.
- Lanțul  $L_6(1,1) = \{1, 2, 3, 6, 5, 4, 1\}$  este un ciclu elementar deoarece nu se repetă nici un nod.

#### 3. Un circuit electric poate fi reprezentat cu ajutorul unui graf orientat $G_{19}$ , în care nodurile rețelei sunt nodurile grafului, iar sensul arcelor este dat de sensul ales pentru curentul electric – figura 20.

Un ochi al rețelei electrice reprezintă un ciclu în graful orientat. Fiecare arc  $k$  are asociate trei mărimi: rezistența  $R_k$ , intensitatea curentului  $I_k$  și tensiunea electromotoare a sursei  $E_k$ . Semnul curentului electric este dat de sensul arcului: dacă arcul intră în nod, curentul electric are semnul plus, iar dacă arcul iese din nod, curentul electric are semnul minus.

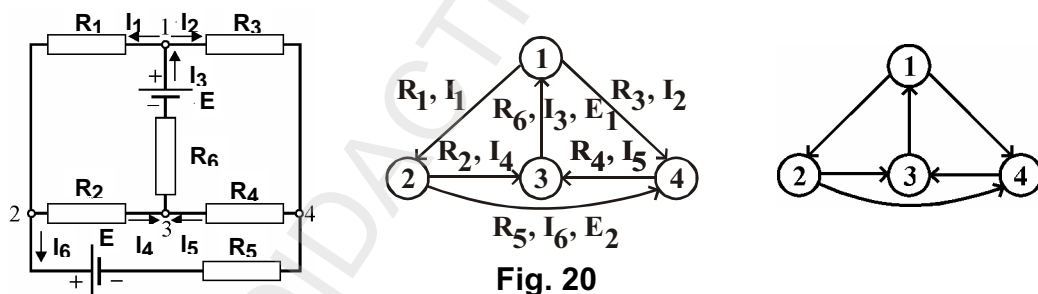


Fig. 20

Pentru **fiecare nod** din graf, se aplică **teorema întâi a lui Kirchhoff**:

$$\sum_{k=1}^p I_k = 0$$

unde  $p$  este suma dintre gradul intern și gradul extern ale nodului.

Pentru **fiecare ciclu** din graf, se aplică **teorema a doua a lui Kirchhoff**:

$$\sum_{k=1}^q E_k = \sum_{k=1}^q I_k \times R_k$$

unde  $q$  este numărul de arce ale ciclului.

### Temă



Pentru circuitul electric din figura 25, se consideră cunoscute următoarele mărimi:  $R_1, R_2, R_3, R_5, R_6, I_1, E_1$  și  $E_2$ . Scrieți un program care să calculeze – în graful circuitului electric – mărimea rezistenței  $R_4$ .

## Teorema 12

**Dacă un graf conține un ciclu, atunci conține și un ciclu elementar.**

## Algoritm pentru determinarea ciclurilor elementare într-un graf

**Afișarea ciclurilor elementare din graf.**

**Algoritmul.** Se generează toate lanțurile elementare care pornesc din nodul  $x$  ( $1 \leq x \leq n$ ), trec prin cel puțin trei noduri și se pot închide cu nodul  $x$ . Pentru generarea acestor lanțuri elementare se folosește **metoda backtracking**, care se va apela pentru fiecare nod  $x$ . Soluția se obține atunci când nodul adăugat în stivă formează o muchie (un arc) cu nodul  $x$  – și stiva conține cel puțin 3 noduri.

**Implementarea algoritmului.** Se citesc, din fișierul text *graf1.txt*, matricea de adiacență a unui graf neorientat, respectiv, din fișierul *graf2.txt*, matricea de adiacență a unui graf orientat. Se caută toate ciclurile elementare care există în graf și se afișează. Dacă nu există nici un ciclu elementar, se afișează un mesaj. Pentru testarea programelor se folosesc graful neorientat  $G_1$  și graful orientat  $G_8$ .

```
#include <fstream.h>
typedef stiva[100];
int a[20][20], n, k, as, ev, x, este=0;
stiva st;
fstream f("graf1.txt", ios::in);
void citeste()
{ //funcția prin care se citește matricea de adiacență din fișier}
void init()
{ //este identică cu cea de la afișarea lanțurilor elementare}
int succesor()
{ //este identică cu cea de la afișarea lanțurilor elementare}
int valid()
{ //este identică cu cea de la afișarea lanțurilor elementare}
int solutie() {return a[st[k]][x]==1 && k>2;}
//se obține soluția atunci când ultimul nod adăugat este adiacent
//nodului x și în stivă există cel puțin trei noduri
void tipar ()
{int i; este=1;
for (i=1; i<=k; i++) cout<<st[i]<<" "; cout<<x<<endl;}
void bt() { //partea fixa a algoritmului }
void main()
{citeste();
for (x=1; x<=n; x++) {st[1]=x; bt();}
if (!este) cout<<"Nu exista nici un ciclu elementar";}
```


**Temă**

1. Scrieți un program care citește, din fișierul text *graf3.txt*, lista muchiilor unui graf neorientat – și afișează toate ciclurile elementare care trec printr-un nod cu gradul minim.
2. Scrieți un program care citește, din fișierul text *graf3.txt*, lista muchiilor unui graf neorientat și care verifică dacă un șir de  $k$  numere citite de la tastatură reprezintă un **ciclu elementar** pentru graf.
3. Scrieți câte un program care citește, din fișierul text *graf4.txt*, lista arcelor unui graf orientat – și care caută toate ciclurile elementare, de lungime maximă, care trec prin nodul  $x$  – și le afișează. Eticheta nodului se citește de la tastatură. Dacă nu există nici un ciclu elementar care să treacă prin nodul  $x$ , se afișează un mesaj.

### 2.6.7.3. Drumul

**Într-un graf orientat  $G=(X,U)$  se definește un drum ca fiind o succesiune de noduri care au proprietatea că – oricare ar fi două noduri succesive – ele sunt legate printr-un arc.**

Dacă, într-un graf orientat, mulțimea nodurilor este  $X=\{x_1, x_2, \dots, x_n\}$ , iar mulțimea arcelor este  $U=\{u_1, u_2, \dots, u_m\}$ , un drum de la nodul  $d_1$  la nodul  $d_k - D(d_1, d_k)$  – va fi definit prin mulțimea nodurilor  $D(d_1, d_k) = \{d_1, d_2, \dots, d_k\}$ , unde  $d_i \in U$ , pentru orice  $i$  ( $1 \leq i \leq k$ ), iar arcele  $[d_1, d_2], [d_2, d_3], [d_3, d_4], \dots, [d_{k-1}, d_k] \in U$ . Drumul poate fi privit ca un lanț în care parcurgerea de la un nod la altul trebuie să se facă în sensul arcului care leagă nodurile. Dacă există  $D(x_i, x_j)$ , se spune că **nodul  $x_j$  este accesibil din nodul  $x_i$ .**

#### Terminologie:

- **Lungimea unui drum** este dată de numărul de arce care îl compun. În cazul în care arcele au asociate lungimi, lungimea unui drum este dată de suma lungimilor arcelor care îl compun.
- **Drumul de lungime minimă** dintre nodul  $d_i$  și nodul  $d_j - D_{\min}(d_i, d_j)$  – este drumul cu numărul minim de arce din mulțimea nevidă de drumuri  $D(d_i, d_j)$ .
- **Subdrumul** este format dintr-un șir continuu de noduri din drum. De exemplu, pentru lanțul  $D(d_1, d_k) = \{d_1, d_2, \dots, d_i, \dots, d_j, \dots, d_k\}$ ,  $D_s(d_i, d_j)$  definit astfel:  $D_s(d_i, d_j) = \{d_i, d_{i+1}, \dots, d_{j-1}, d_j\}$  – este un subdrum al drumului  $D$ .

**Drumul elementar** este drumul în care nodurile sunt distincte două câte două. **Drumul simplu** este drumul în care arcele sunt distincte două câte două.

#### Exemple:

Pentru graful orientat  $G_{18}$ :

- Lanțul  $L_7(1,6) = \{1, 2, 5, 6\}$  nu este un drum deoarece parcurgerea nu se face în sensul săgeților.
- Lanțul  $L_8(1,6) = \{1, 2, 3, 6, 3, 6\} = D_1$  este un drum neelementar, deoarece se repetă eticheta nodurilor 3 și 6 – și compus, deoarece prin arcul  $[3,6]$  s-a trecut de două ori.
- Lanțul  $L_9(1,6) = \{1, 2, 3, 7, 6\} = D_2$  este un drum elementar, deoarece nu se repetă nici un nod.
- Lanțul  $L_9(1,6) = \{1, 2, 4, 5, 2, 3, 6\} = D_3$  este un drum simplu, deoarece nici un arc nu a fost parcurs de două ori, dar este un drum neelementar, deoarece se repetă nodul cu eticheta 2.

#### Teorema 13

**Dacă un graf neorientat conține un drum între două noduri,  $x$  și  $y$ , atunci, conține și un drum elementar, între nodurile  $x$  și  $y$ .**

### Algoritm pentru determinarea drumurilor elementare într-un graf orientat

#### Afișarea drumurilor elementare din graf.

**Algoritmul.** Se va folosi algoritmul pentru determinarea tuturor lanțurilor elementare din graf, la care elementul soluției generat în stivă, la nivelul  $k$  ( $k > 1$ ), va fi considerat valid numai dacă trecerea de la nodul de la nivelul  $k-1$  la nodul de la nivelul  $k$  se face, în graf, în sensul arcului.

**Implementarea algoritmului.** Se citește, din fișierul *graf2.txt*, matricea de adiacență a unui graf orientat. Se caută toate drumurile elementare care există în graf – și se afișează. Dacă nu există nici un drum elementar, se afișează un mesaj. Pentru testarea programului se folosește graful orientat  $G_8$ .

```
#include <fstream.h>
typedef stiva[100];
int a[20][20],n,k,as,ev,x,este=0;
stiva st;
fstream f("graf2.txt",ios::in);
void citeste()
{//funcția prin care se citește matricea de adiacență din fișier}
void init()
{//este identică cu cea de la afișarea lanțurilor elementare}
int succesor()
{//este identică cu cea de la afișarea lanțurilor elementare}
int valid()
{if(k>1) if (a[st[k-1]][st[k]]==0) return 0;
 for (int i=1;i<k;i++) if (st[k]==st[i]) return 0;
 return 1;}
int solutie() {return st[k]==y;}
void tipar ()
{int i; este=1;
 for (i=1;i<=k;i++) cout<<st[i]<<" "; cout<<x<<endl;}
void bt() {//partea fixa a algoritmului }
void main()
{citeste();
 for (x=1;x<=n;x++)
 for (y=1;y<=n;y++) if (x!=y) {st[1]=x; bt();}
 if (!este) cout<<"Nu exista";}
```

### Temă



Scriveți un program care citește, din fișierul text *graf4.txt*, lista muchiilor unui graf orientat – și care:

- Verifică dacă un șir de  $k$  numere citite de la tastatură reprezintă un **drum elementar** pentru graf.
- Caută toate drumurile elementare cu lungimea cea mai mică dintre două noduri  $x$  și  $y$  și le afișează. Etichetele nodurilor se citesc de la tastatură. Dacă nu există nici un drum elementar, să se afișeze un mesaj.

## 2.6.7.4. Circuitul

**Într-un graf orientat un drum care are toate arcele distincte două câte două și extremități care coincid se numește circuit.**

Circuitul este un **drum simplu** (arcele sunt distincte două câte două ( $[d_1, d_2] \neq [d_2, d_3]; [d_1, d_2] \neq [d_3, d_4]; [d_1, d_2] \neq [d_4, d_5]; \dots; [d_1, d_2] \neq [d_{k-1}, d_k]; \dots; [d_{k-2}, d_{k-1}] \neq [d_{k-1}, d_k]$ ) în care extremitățile coincid ( $d_1 = d_k$ ).

**Circuitul elementar** este circuitul în care toate nodurile sunt distincte două câte două, cu excepția primului și a ultimului – care coincid.

**Exemple** – Pentru graful orientat  $G_{18}$ :

- Lanțul  $L_{10}(1,1) = \{1, 2, 3, 6, 3, 6, 3, 7, 6, 5, 4, 1\}$  nu este un circuit, deoarece arcele  $[3,6]$  și  $[6,3]$  au fost parcurse de două ori.
- Lanțul  $L_{11}(1,1) = \{1, 2, 3, 6, 3, 7, 6, 5, 2, 4, 1\} = C_1$  este un circuit neelementar, deoarece se repetă eticheta nodurilor 3 și 6.
- Lanțul  $L_{12}(1,1) = \{1, 2, 3, 7, 6, 5, 4, 1\} = C_2$  este un circuit elementar, deoarece nu se repetă eticheta niciunui nod.

## Teorema 14

**Dacă un graf conține un circuit, atunci conține și un circuit elementar.**

## Algoritm pentru determinarea circuitelor elementare într-un graf orientat

**Afișarea circuitelor elementare din graf.**

**Algoritmul.** Se folosește același algoritm ca și în cazul determinării ciclurilor elementare într-un graf orientat, cu deosebirea că se ține cont de orientarea arcului care leagă două noduri.

```
#include <fstream.h>
typedef stiva[100];
int a[20][20], n, k, as, ev, x, este=0;
stiva st;
fstream f("graf2.txt", ios::in);
void citeste()
//funcția prin care se citește matricea de adiacență din fișier}
void init()
//este identică cu cea de la afișarea lanțurilor elementare}
int succesor()
//este identică cu cea de la afișarea lanțurilor elementare}
int valid()
{if(k>1) if (a[st[k-1]][st[k]]==0) return 0;
 for (int i=1; i<k; i++) if (st[k]==st[i]) return 0;
return 1;}
int solutie() {return a[st[k]][x]==1 && k>2;}
//se obține soluția atunci când ultimul nod adăugat este
//predecesor nodului x și în stivă există cel puțin trei noduri
void tipar ()
{int i; este=1;
 for (i=1; i<=k; i++) cout<<st[i]<<" "; cout<<x<<endl;}
void bt() {//partea fixa a algoritmului }
void main()
{citeste();
 for (x=1; x<=n; x++) {st[1]=x; bt();}
 if (!este) cout<<"Nu exista circuite";}
```


**Temă**

Scrieți un program care citește, din fișierul text *graf4.txt*, lista arcelor unui graf orientat – și care:

- a. Caută toate circuitele elementare care există în graf și le afișează. Dacă nu există nici un circuit elementar, se afișează un mesaj.
- b. Caută toate circuitele elementare de lungime maximă care trec printr-un nod **x** și le afișează. Eticheta nodului se citește de la tastatură. Dacă nu există nici un circuit elementar care să treacă prin nodul **x**, se afișează un mesaj.
- c. Caută toate circuitele elementare care trec numai prin noduri care au gradul intern egal cu gradul extern – și le afișează. Dacă nu există nici un circuit elementar, se afișează un mesaj.

## 2.6.7.5. Graful conex

Un graf **G** se numește **graf conex** dacă are proprietatea că, pentru orice pereche de noduri diferite între ele, există un lanț care să le lege.

Altfel spus, un graf este conex dacă, pentru orice pereche de noduri  $\{x_i, x_j\}$ , care au proprietatea  $x_i \neq x_j$  există un lanț de la  $x_i$  la  $x_j$ .

### Exemple:

1. Graful neorientat  $G_{17} = (X_{17}, U_{17})$  definit anterior este un graf conex, deoarece oricare ar fi două noduri din mulțimea  $X_x$ , ele pot fi legate printr-un lanț. De exemplu,  $(1,2) \Rightarrow \{1,2\}$  sau  $\{1,5,2\}$ ;  $(1,3) \Rightarrow \{1,2,3\}$  sau  $\{1,6,3\}$ ;  $(1,3) \Rightarrow \{1,2,3\}$  sau  $\{1,6,3\}$ ;  $(1,4) \Rightarrow \{1,2,3,4\}$  sau  $\{1,6,7,4\}$ ;  $(1,5) \Rightarrow \{1,5\}$  sau  $\{1,2,6,5\}$  etc.
2. Graful orientat  $G_{18} = (X_{18}, U_{18})$  definit anterior este un graf conex, deoarece oricare ar fi două noduri din mulțimea  $X_x$ , ele pot fi legate printr-un lanț. De exemplu,  $(1,2) \Rightarrow \{1,2\}$  sau  $\{1,4,2\}$ ;  $(1,3) \Rightarrow \{1,2,3\}$  sau  $\{1,2,5,6,3\}$ ;  $(1,3) \Rightarrow \{1,2,3\}$  sau  $\{1,2,5,6,3\}$ ;  $(1,4) \Rightarrow \{1,4\}$  sau  $\{1,2,5,4\}$ ;  $(1,5) \Rightarrow \{1,2,5\}$  sau  $\{1,4,5\}$  etc.
3. Graful neorientat  $G_{20} = (X_{20}, U_{20})$  – figura 21 – definit astfel:

→ mulțimea nodurilor este  $X_{20} = \{1,2,3,4,5,6,7,8\}$ .

→ mulțimea muchiilor este  $U_{20} = \{[1,2], [1,4], [2,3], [2,4], [3,4], [5,6], [5,7]\}$ .

nu este conex, deoarece nu există nici un lanț între un nod din mulțimea  $C_1 = \{1,2,3,4\}$  și un nod din mulțimea  $C_2 = \{5,6,7\}$ , sau din mulțimea  $C_3 = \{8\}$ .

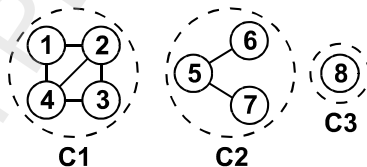


Fig. 21

Dacă un graf  $G=(X,U)$  nu este conex, se poate defini un **subgraf conex** al grafului  $G$ , adică se poate defini o mulțime  $X' \subset X$  care să inducă subgraful  $G'=(X',U')$ , ce are proprietatea că este conex.

**Dacă un graf  $G=(X,U)$  nu este conex, se numește componentă conexă a grafului un subgraf conex al său  $C=(X',U')$ , maximal în raport cu această proprietate (conține numărul maxim de noduri din  $G$  care au proprietatea că sunt legate cu un lanț).**

Altfel spus, subgraful conex  $C$  este o componentă conexă a grafului dacă are proprietatea că nu există nici un lanț, al grafului  $G$ , care să unească un nod  $x_i$  al subgrafului  $C$  ( $x_i \in X'$ ) cu un nod  $x_j$  care nu aparține subgrafului  $C$  ( $x_j \in X - X'$ ).

**Observație:** Un graf conex are o singură componentă conexă (graful însuși).

**Exemplu** – În graful neorientat  $G_{20}$  definit anterior, fiecare dintre cele trei mulțimi de noduri,  $C_1$ ,  $C_2$  și  $C_3$ , induce câte o componentă conexă.

Dacă un graf este definit prin mulțimea nodurilor și mulțimea muchiilor,

→ mulțimea nodurilor este  $X_{23} = \{1,2,3,4,5,6,7,8\}$ .

→ mulțimea muchiilor este  $U_{23} = \{[1,2], [1,4], [2,3], [2,4], [3,4], [5,6], [5,7]\}$ .

pentru identificarea componentelor conexe, procedați astfel:

**PAS1.** Se formează prima componentă conexă, pornind de la prima muchie. Se scriu nodurile incidente cu muchia în prima componentă conexă –  $C_1 = \{1,2\}$  – și se marchează muchia: [1,2]. Componenta conexă  $C_1$  este componenta curentă.

**PAS2.** Se parcurg celelalte muchii. Dacă se găsește o muchie nemarcată, care conține noduri din componenta conexă curentă, se adaugă nodurile la componentă:  $C_1 = \{1,2,4,3\}$ , și se marchează muchia: [1,4], [2,3], [2,4], [3,4].

**PAS3.** Dacă mai există muchii nemarcate, se identifică prima muchie nemarcată și se formează următoarea componentă conexă, scriind nodurile incidente cu muchia –  $C_2 = \{5,6\}$  – și se marchează muchia: [5,6]. Componenta conexă  $C_2$  este componenta curentă.

**PAS4.** Se parcurg celelalte muchii. Dacă se găsește o muchie nemarcată care conține noduri din componenta conexă curentă, se adaugă nodurile la componentă:  $C_2 = \{5,6,7\}$  și se marchează muchia: [5,7].



- PAS5.** Se execută Pasul 3 și Pasul 4 până se marchează toate muchiile, identificând următoarele componente conexe.
- PAS6.** Se verifică dacă toate nodurile din mulțimea nodurilor se regăsesc într-o componentă conexă. Dacă există noduri care nu aparțin unei componente conexe, acestea sunt noduri izolate, și vor forma, fiecare dintre ele, o componentă conexă.  $C_3=\{8\}$ .

### Teorema 15

**Numărul minim de muchii**  $m_{\min}$  necesare pentru ca un graf neorientat, cu  $n$  noduri, să fie conex este  $n-1$ .

Altfel spus, într-un graf neorientat conex cu  $m$  muchii și  $n$  noduri:  $m \geq n-1$ .

**Demonstrație.** Notăm cele două propoziții, astfel:

- (1) – Graful  $G$  neorientat, cu  $n$  noduri, are  $n-1$  muchii și este conex.
- (2) – Graful  $G$  neorientat, cu  $n$  noduri, este conex și minimal cu această proprietate (dacă se înlătură orice muchie din graf, el devine neconex).

Trebuie să demonstrăm că  $(1) \Rightarrow (2)$  – Se folosește principiul inducției matematice (se notează cu  $P_i$  propoziția  $i$ ):

**$P_1$**  – Graful  $G_1$ , cu un nod și nici o muchie ( $m=1-1=0$ ), este conex și este minimal cu această proprietate (nu mai există muchii care să fie eliminate).

**$P_2$**  – Graful  $G_2$ , cu două noduri și o muchie ( $m=2-1=1$ ), este conex (muchia nu poate lega decât cele două noduri) și este minimal cu această proprietate (prin eliminarea muchiei, nodurile sunt izolate obținându-se două componente conexe).

**$P_3$**  – Graful  $G_3$ , cu trei noduri și două muchii ( $m=3-1=2$ ), este conex (cele două muchii sunt incidente; unul dintre noduri este adiacent cu cele două muchii – și prin el va trece lanțul care leagă celelalte două noduri) și este minimal cu această proprietate (prin eliminarea unei muchii, unul dintre noduri este izolat, obținându-se două componente conexe).

**$P_n$**  – Graful  $G_n$ , cu  $n$  noduri și  $n-1$  muchii, este conex și minimal cu această proprietate (se presupune adevărată).

**$P_{n+1}$**  – Considerând propoziția  $P_n$  adevărată, trebuie să demonstrăm că graful  $G_{n+1}$ , cu  $n+1$  noduri și  $n+2$  muchii, este conex și minimal cu această proprietate. Prin adăugarea unui nod și a unei muchii la graful  $G_n$ , se poate obține un graf conex. Nodul adăugat se leagă cu muchia nouă de graful  $G_n$  (și, implicit, la lanțurile din acest graf), obținându-se un nou graf conex. Presupunem că graful obținut nu este minimal cu această proprietate. Înseamnă că, prin eliminarea unei muchii, se obține un graf conex. Dacă eliminăm muchia adăugată, se izolează nodul  $n+1$ , obținându-se două componente conexe. Dacă se elimină o muchie din graful  $G_n$ , se obțin două componente conexe, deoarece graful  $G_n$  este minimal cu această proprietate. Rezultă că graful  $G_n$  este minimal cu această proprietate.

**Exemple** de grafuri conexe cu  $n$  noduri și număr minim de muchii:

→ Fiecare nod este legat de alte două, cu excepția a două noduri terminale. Există  $n-2$  noduri cu gradul 2 și două noduri cu gradul 1:

$$m_{\min} = \frac{2 \times (n-2) + 2 \times 1}{2} = n-1$$

→ Un nod este legat de celelalte  $n-1$  noduri. Există un nod cu gradul  $n-1$ , și  $n-1$  noduri cu gradul 1:

$$m_{\min} = \frac{1 \times (n-1) + (n-1) \times 1}{2} = n-1$$

**Propoziția 3.** Dacă un graf cu  $n$  noduri are  $p$  componente conexe, atunci numărul minim de muchii care trebuie adăugate, ca să devină conex, este  $p-1$ .

**Demonstrație.** Putem considera fiecare componentă conexă ca un nod al unui graf cu  $p$  noduri. Numărul minim de muchii necesare pentru ca acest graf să fie conex este  $p-1$ .

**Propoziția 4.** Dacă un graf conex cu  $n$  noduri are  $n-1$  muchii, atunci orice pereche de noduri este legată printr-un lanț, și numai unul.

**Demonstrație** – prin reducere la absurd. Graful  $G$  fiind conex, înseamnă că există cel puțin un lanț care leagă oricare două noduri,  $x$  și  $y$ . Presupunem că există cel puțin două lanțuri între nodurile  $x$  și  $y$ :  $L_1$  și  $L_2$ . Înseamnă că, suprimând o muchie din lanțul al doilea, graful rămâne conex, deoarece nodurile  $x$  și  $y$  vor fi legate prin lanțul  $L_1$ . Înseamnă că un graf cu  $n$  noduri și  $n-2$  muchii este conex, ceea ce contrazice Teorema 15. Rezultă că cele două noduri,  $x$  și  $y$ , nu sunt legate decât printr-un singur lanț.

**Propoziția 5.** Dacă un graf neorientat cu  $n$  noduri și  $m$  muchii este conex, numărul maxim de muchii care se pot elimina pentru a obține un **graf parțial conex** este:  $m-n+1$ .

**Demonstrație.** Deoarece numărul minim de muchii necesare pentru ca un graf să fie conex este  $n-1$ , atunci din graf se pot elimina restul muchiilor:  $m-n+1$ .

### Teorema 16

Un **graf neorientat conex**, cu  $n$  noduri și  $n-1$  muchii, este **aciclic** și maximal cu această proprietate.

**Demonstrație:** Notăm cele două propoziții, astfel:

- (1) – Graful  $G$  neorientat, cu  $n$  noduri, este conex și are  $n-1$  muchii.
- (2) – Graful  $G$  neorientat, cu  $n$  noduri, este aciclic și maximal cu această proprietate.

Trebuie să demonstrăm că  $(1) \Rightarrow (2)$ .

- 1) **Este aciclic** – prin reducere la absurd. Presupunem că acest graf conține cel puțin un ciclu,  $C = \{x_1, x_2, x_3, \dots, x_i, x_1\}$ . Înseamnă că, dacă vom înlătura din graf muchia  $[x_1, x_2]$ , se obține un graf conex – nodul va fi legat de toate celelalte noduri din ciclu prin lanțul  $L = \{x_2, x_3, \dots, x_i, x_1\}$  – ceea ce contrazice teorema anterioară, că un graf cu  $n$  noduri și  $n-1$  muchii este conex și minimal cu această proprietate.
- 2) **Este maximal cu această proprietate** (dacă se adaugă o nouă muchie, graful nu mai este aciclic) – prin reducere la absurd. Presupunem că, prin adăugarea unei muchii oarecare  $[x, y]$ , se obține un graf aciclic. Graful fiind conex, înseamnă că între nodurile  $x$  și  $y$  există un lanț  $L(x, y) = \{x, \dots, z, \dots, y\}$ , iar prin adăugarea muchiei  $[x, y]$  lanțul se închide, formând un ciclu  $C = \{x, \dots, z, \dots, y, x\}$  – ceea ce contrazice presupunerea că graful este aciclic.

**Propoziția 6.** Dacă un graf neorientat conex are  $n$  noduri și  $m$  muchii, numărul de muchii care trebuie eliminate, pentru a obține un **graf parțial conex aciclic**, este egal cu  $m-n+1$ .

**Demonstrație.** Din Propoziția 5, rezultă că – înlăturând din graf  $m-n+1$  muchii – se obține un graf parțial conex, iar din Teorema 16 rezultă că acest graf este aciclic. Graful fiind conex, înseamnă că între oricare două noduri,  $x_i$  și  $x_j$ , există un lanț elementar și numai unul (Propoziția 4). Orice nouă muchie  $[x_i, x_j]$  adăugată va forma cu acest lanț un ciclu elementar.

**Propoziția 7.** Dacă un graf are  $n$  noduri,  $m$  muchii și  $p$  componente conexe, numărul de muchii care trebuie eliminate, pentru a obține un **graf parțial aciclic**, este egal cu  $m-n+p$ .

**Demonstrație.** Fiecare componentă conexă  $i$  ( $1 \leq i \leq p$ ) va avea  $n_i$  noduri și  $m_i$  muchii, iar:

$$\sum_{i=1}^p n_i = n \text{ și } \sum_{i=1}^p m_i = m.$$

Numărul de muchii care trebuie eliminate din componenta conexă  $i$ , pentru a obține un graf parțial conex aciclic, este egal cu  $m_i - n_i + 1$  (Propoziția 5). Rezultă că numărul total de muchii care trebuie eliminate din graf, pentru a obține graf parțial conex aciclic, este egal cu:

$$\sum_{i=1}^p (m_i - n_i + 1) = \sum_{i=1}^p m_i - \sum_{i=1}^p n_i + p = m - n + p$$

**Propoziția 8.** Pentru a obține, dintr-un graf neorientat conex, două componente conexe, numărul minim de muchii care trebuie înlăturate  $m_{\min}$  este egal cu gradul minim din graf:  $m_{\min} = \text{grad}_{\min}$ .

**Demonstrație.** Cele două componente conexe ale unui graf neorientat conex  $G=(X,U)$  se obțin astfel:

→ Componenta conexă  $C_1=(X_1,U_1)$  se formează dintr-un nod izolat  $x_i$  – pentru a elimina un număr minim de muchii, nodul se alege dintre nodurile care au gradul minim:  $X_1=\{x_i\}$  și  $U_1=\emptyset$ .

→ Componenta conexă  $C_2=(X_2,U_2)$  se formează din restul nodurilor din graf:  $X_2=X-\{x_i\}$  și  $\text{card}(U_1)=m-\text{grad}_{\min}$ .

### 2.6.7.6. Graful tare conex

**Un graf orientat  $G$  se numește graf tare conex dacă are proprietatea că, pentru orice pereche de noduri diferite între ele, există un drum care să le lege.**

Altfel spus, un graf orientat este tare conex dacă – pentru orice pereche de noduri  $\{x_i, x_j\}$ , care au proprietatea  $x_i \neq x_j$  – există un drum de la  $x_i$  la  $x_j$ .

**Exemple:**

1. Graful orientat  $G_{18} = (X_{18}, U_{18})$ , definit anterior, este un graf tare conex, deoarece există un circuit elementar care trece prin toate nodurile grafului:  $\{1, 2, 3, 7, 6, 5, 4, 1\}$ ; altfel spus, oricare ar fi două noduri din mulțimea  $X_{20}$ , ele pot fi legate printr-un drum.

2. Graful orientat  $G_{21} = (X_{21}, U_{21})$  – figura 22 – definit astfel:

→ mulțimea nodurilor este  $X_{21}=\{1, 2, 3, 4, 5, 6\}$ .

→ mulțimea arcelor este  $U_{21}=\{[1, 2], [1, 4], [2, 3], [3, 1], [4, 5], [5, 4]\}$ .

nu este conex, deoarece nu există nici un drum între un nod din mulțimea  $C_2=\{4, 5\}$  și un nod din mulțimea  $C_1=\{1, 2, 3\}$ , sau din mulțimea  $C_3=\{6\}$ .

Dacă un graf orientat  $G=(X,U)$  nu este tare conex, se poate defini un **subgraf tare conex** al grafului  $G$ , adică se poate defini o mulțime  $X' \subset X$  care să inducă subgraful  $G'=(X',U')$ , ce are proprietatea că este tare conex.

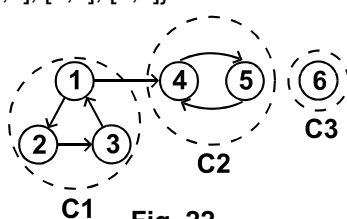


Fig. 22

**Dacă un graf  $G=(X,U)$  nu este conex, se numește componentă tare conexă a grafului un subgraf conex  $C=(X',U')$  al său, maximal în raport cu această proprietate (conține numărul maxim de noduri din  $G$  care au proprietatea că sunt legate printr-un drum).**

Altfel spus, subgraful tare conex  $C$  este o componentă tare conexă a grafului dacă are proprietatea că nu există nici un drum al grafului  $G$  care să unească un nod  $x_i$  al subgrafului  $C$  ( $x_i \in X'$ ) cu un nod  $x_j$  care nu aparține subgrafului  $C$  ( $x_j \in X-X'$ ).

**Exemplu** – În graful orientat  $G_{21}$  definit anterior, fiecare dintre cele trei mulțimi de noduri –  $C_1$ ,  $C_2$  și  $C_3$  – induce câte o componentă tare conexă.

**Observație:** Un graf tare conex are o singură componentă tare conexă (graful însuși).

**Terminologie:**

→ **Subgraful predecesorilor unui nod** este format din acel nod și din mulțimea nodurilor din care este accesibil nodul.

→ **Subgraful succesorilor unui nod** este format din acel nod și din mulțimea nodurilor care sunt accesibile din el.

**Observație:** Componenta tare conexă din care face parte un nod este dată de intersecția dintre subgraful predecesorilor și subgraful succesorilor acelui nod.

Dacă un graf este definit prin mulțimea nodurilor și mulțimea muchiilor,

→ **mulțimea nodurilor** este  $X_{24}=\{1,2,3,4,5,6\}$ .

→ **mulțimea arcelor** este  $U_{24}=\{[1,2], [1,4], [2,3], [3,1], [4,5], [5,4]\}$ .

pentru identificarea componentelor tare conexe procedați astfel:

**PAS1.** Se identifică subgraful succesorilor primului nod din primul arc (în exemplu, nodul 1), folosind algoritmul de determinare a unei componente conexe:  $S_1=\{1,2,3,4,5\}$ .

**PAS2.** Se identifică subgraful predecesorilor primului nod din primul arc (în exemplu, nodul 1), folosind algoritmul de determinare a unei componente conexe, în care se iau în calcul numai nodurile care apar în arc în a doua poziție:  $P_1=\{1,2,3\}$ .

**PAS3.** Se determină componenta tare conexă – prin intersecția celor două mulțimi de noduri:  $C_1=S_1 \cap P_1=\{1,2,3\}$ .

**PAS4.** Se identifică, în mulțimea arcelor, primul nod care nu face parte din componenta tare conexă evidențiată anterior (în exemplu, nodul 4) și se reiau **Pasul 1**, **Pasul 2** și **Pasul 3**, pentru a determina subgraful succesorilor, respectiv subgraful predecesorilor nodului și apoi următoarea componentă conexă, prin intersecția celor două mulțimi (în exemplu, nodul 4 și  $S_2=\{1,4,5\}$ ,  $P_2=\{1,2,3,4,5\}$  și  $C_2=S_2 \cap P_2=\{4,5\}$ ). Se repetă acest pas până când nu se mai identifică în mulțimea arcelor niciun nod care să nu aparțină unei componente tare conexe.

**PAS5.** Se verifică dacă toate nodurile din mulțimea nodurilor se regăsesc într-o componentă tare conexă. Dacă există noduri care nu aparțin unei componente tare conexe, acestea sunt noduri izolate – și vor forma, fiecare dintre ele o componentă conexă.  $C_3=\{6\}$ .

## Algoritmi pentru determinarea conexității grafurilor

### 1. Determinarea componentelor conexe într-un graf.

**Algoritmul.** O componentă conexă este formată din toate nodurile între care există un lanț elementar. Deoarece un nod nu poate să aparțină decât unei componente conexe, se va folosi un vector cu  $n$  elemente – pentru a memora nodurile, care au fost deja înregistrate într-o componentă conexă. Inițial, elementele vectorului au valoarea 0 (nici un nod nu a fost înregistrat într-o componentă conexă), iar dacă un nod  $x$  va fi adăugat la o componentă conexă, valoarea elementului  $x$  din vector va fi 1. Pentru a **verifica dacă există un lanț elementar între două noduri  $x$  și  $y$** , se folosește **metoda backtracking**.

**Observație.** Acest algoritm poate fi folosit atât în grafurile neorientate, cât și în grafurile orientate. Mai poate fi folosit și pentru a verifica dacă un graf este conex: fie se verifică dacă între nodul 1 și fiecare dintre celelalte noduri ale grafului există un lanț elementar, fie se verifică dacă prima componentă conexă obținută conține toate nodurile grafului.

**Implementarea algoritmului.** Se citește, din fișierul *graf19.txt*, matricea de adiacență a unui graf neorientat (variantea din fișierul *graf20.txt* – matricea de adiacență a unui graf orientat) și se afișează componentele conexe. În vectorul  $v$  se memorează nodurile care au fost deja înregistrate într-o componentă conexă. Variabila **este** se folosește pentru a verifica dacă s-a găsit un lanț elementar între două noduri (are valoarea 0 – *False*, dacă nu s-a găsit un lanț elementar). Variabila **m** se folosește pentru a număra componentele conexe identificate. Funcția **citeste()** se folosește pentru a citi matricea de adiacență în fișier. Funcția **componente()** se folosește pentru a afișa componentele conexe ale grafului: pentru fiecare nod  $x$  ( $1 \leq x \leq n$ ) care nu a fost afișat într-o componentă conexă ( $v[x]=0$ ), se caută toate nodurile  $y$  ( $1 \leq y \leq n$  și  $y \neq x$ ) care sunt legate cu un lanț elementar de nodul  $x$ . Pentru un nod  $y$  găsit, se marchează în vectorul  $v$  faptul că a fost adăugat la o

componentă conexă ( $v[y]=1$ ). Pentru testarea programului se folosesc graful neorientat  $G_4$  și graful orientat  $G_{20}$ .

```
#include <fstream.h>
typedef stiva[100];
int a[20][20], n, k, as, ev, x, y, v[20], este=0;
stiva st;
fstream f("gr8af19.txt", ios::in);
void citeste()
//funcția prin care se citește matricea de adiacență din fișier}
void init()
//este identică cu cea de la afișarea lanțurilor elementare}
int succesor()
//este identică cu cea de la afișarea lanțurilor elementare}
int valid()
//este identică cu cea de la afișarea lanțurilor elementare}
int solutie() {return st[k]==y;}
//se obține soluția atunci când ultimul nod adăugat în stivă este y
void tipar() {este=1;}
//dacă există soluție (un lanț între nodurile x și y),
// variabila este va avea valoarea 1
void bt()
//este identică cu cea de la afișarea lanțurilor elementare }
void componente()
{int m=0;
 for(x=1; x<=n; x++)
 if (v[x]==0)
 {m++; st[1]=x; v[x]=1;
 cout<<"Componenta conexa "<<m<<": "<<x<<" ";
 for(y=1; y<=n; y++)
 if (x!=y) {este=0; bt();
 if (este) {v[y]=1; cout<<y<<" ";}}
 cout<<endl;}}
void main() {citeste(); componente();}
```

## 2. Determinarea componentelor tare conexe într-un graf orientat.

**Algoritmul.** O componentă tare conexă este formată din toate nodurile  $i$  și  $j$  între care există un drum elementar de la nodul  $i$  la nodul  $j$ , dar și de la nodul  $j$  la nodul  $i$ . Se folosește un vector cu  $n$  elemente, pentru a memora nodurile care au fost deja înregistrate într-o componentă tare conexă. Inițial, elementele vectorului au valoarea 0 (nici un nod nu a fost înregistrat într-o componentă conexă), iar dacă un nod  $i$  va fi adăugat la o componentă conexă, valoarea elementului  $i$  din vector va fi 1. Pentru a **verifica dacă există un drum elementar de la nodul  $i$  la nodul  $j$** , se folosește metoda **backtracking**.

**Implementarea algoritmului.** Se citește, din fișierul *graf20.txt*, matricea de adiacență a unui graf orientat – și se determină dacă graful este tare conex. Dacă nu este, se afișează componentele tare conexe. Variabila **este1** se folosește pentru a verifica dacă s-a găsit un drum elementar de la nodul  $i$  la nodul  $j$  (are valoarea 0 – *False*, dacă nu s-a găsit un drum elementar), variabila **este2** se folosește pentru a verifica dacă s-a găsit un drum elementar de la nodul  $j$  la nodul  $i$  (are valoarea 0 – *False*, dacă nu s-a găsit un drum elementar), iar variabila **este** se folosește pentru a verifica dacă s-a găsit un drum elementar de la nodul  $x$  la nodul  $y$  (are valoarea 0 – *False*, dacă nu s-a găsit un drum

elementar). În vectorul  $\mathbf{v}$  se memorează nodurile care au fost deja înregistrate într-o componentă tare conexă. Variabila  $\mathbf{m}$  se folosește pentru a număra componentele conexe identificate. Funcția `citeste()` se folosește pentru a citi matricea de adiacență din fișier. Funcția `componente()` se folosește pentru a afișa componentele tare conexe ale grafului: pentru fiecare nod  $i$  ( $1 \leq i \leq n$ ) care nu a fost afișat într-o componentă conexă ( $\mathbf{v}[i]=0$ ), se caută toate nodurile  $j$  ( $1 \leq j \leq n$  și  $j \neq i$ ) care sunt legate cu un drum elementar care pornește din nodul  $i$ , dar și de la care pornește un drum care ajunge în nodul  $i$ . Pentru un nod  $j$  găsit, se marchează în vectorul  $\mathbf{v}$  faptul că a fost adăugat la o componentă tare conexă ( $\mathbf{v}[j]=1$ ). Pentru testarea programului se folosește graful orientat  $G_{20}$ .

```
#include <fstream.h>
typedef stiva[100];
int a[20][20], n, k, as, ev, x, y, v[20], este1, este2, este=0;
stiva st;
fstream f("graf19.txt", ios::in);
void citeste()
{//funcția prin care se citește matricea de adiacență din fișier}
void init()
{//este identică cu cea de la afișarea drumurilor elementare}
int succes()
{//este identică cu cea de la afișarea drumurilor elementare}
int valid()
{//este identică cu cea de la afișarea drumurilor elementare}
int solutie()
{return st[k]==y;}
//se obține soluția atunci când ultimul nod adăugat în stivă este y
void tipar() {este=1;}
//dacă există soluție (un drum între nodurile x și y),
// variabila este va avea valoarea 1
void bt()
{//este identică cu cea de la afișarea drumurilor elementare }
void componente()
{int i, j, m=0;
 for(i=1; i<=n; i++)
 if (v[i]==0)
 {m++; v[i]=1; cout<<"Componenta conexa "<<m<<": "<<i<<" ";
 for(j=1; j<=n; j++)
 if (j!=i)
 {x=i; y=j; st[1]=x; este=0; bt(); este1=este;
 x=j; y=i; st[1]=x; este=0; bt(); este2=este;
 if (este1 && este2) {v[j]=1; cout<<j<<" ";}}
 cout<<endl;}}
void main() {citeste(); componente();}
```

### Temă



1. Scrieți un program care citește, din fișierul text *graf3.txt* (*graf4.txt*), lista muchiilor (arcelor) unui graf neorientat (orientat) – și care verifică dacă graful este conex și afișează un mesaj de informare.
2. Scrieți un program care citește, din fișierul text *graf14.txt*, lista de adiacență a unui graf orientat – și care verifică dacă graful este tare conex și afișează un mesaj de informare.

## 2.6.8. Parcurgerea grafului

**Parcurgerea grafului** reprezintă operația prin care sunt examinate în mod sistematic nodurile sale, pornind de la un nod dat,  $i$ , astfel încât fiecare nod accesibil din nodul  $i$  pe muchiile (arcele) adiacente să fie atins o singură dată. **Vecinii unui nod** sunt reprezentați de toate nodurile adiacente lui și accesibile din el.

**Vizitarea** sau **traversarea** unui graf este operația prin care se parcurge graful trecându-se de la un nod  $i$  (nodul curent) la nodurile vecine lui, într-o anumită ordine, în vederea prelucrării informațiilor asociate nodurilor. Pentru parcurgerea grafurilor, există următoarele **două metode**:

1. **Metoda de parcurgere „în lățime” – Breadth First (BF)**. Se vizitează mai întâi un **nod inițial**  $i$ , apoi vecinii acestuia, apoi vecinii nevizitați ai acestora, și așa mai departe – până când se parcurg toate nodurile grafului.
2. **Metoda de parcurgere „în adâncime” – Depth First (DF)**. Se vizitează mai întâi un **nod inițial**  $i$ , după care se parcurge primul dintre vecinii săi nevizitați, de exemplu  $j$ , după care se trece la primul vecin nevizitat al lui  $j$ , și așa mai departe – până când se parcurge în adâncime ramura respectivă. Când s-a ajuns la capătul ei, se revine la nodul din care s-a plecat ultima dată, și se parcurge următorul său vecin nevizitat.

### Exemplu

**G<sub>22</sub>** Pentru graful neorientat  $G_{22} = (X_{22}, U_{22})$  – figura 23 – definit astfel:

→ **mulțimea nodurilor** este  $X_{22} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ .

→ **mulțimea muchiilor** este  $U_{22} = \{[1, 2], [1, 3], [1, 4], [2, 5], [3, 5], [3, 6], [3, 7], [4, 7], [5, 8], [6, 8], [6, 9]\}$ .

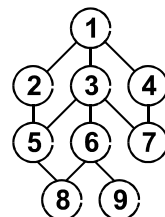


Fig. 23

Graful  $G_{22}$  poate fi parcurs astfel, pornind din nodul 1:

1. În cazul metodei de parcurgere **BF** – „în lățime”, vor fi parcurse pe rând nodurile:  
**1, 2, 3, 4, 5, 6, 7, 8, 9**
2. În cazul metodei de parcurgere **DF** – „în adâncime”, vor fi parcurse pe rând nodurile:  
**1, 2, 5, 3, 6, 8, 9, 7, 4**

### 2.6.8.1. Parcurgerea în lățime – Breadth First

Pentru această metodă de parcurgere a unui graf cu  $n$  noduri, se folosesc următoarele structuri de date și date elementare:

1. **Variabilele de memorie:**  $n$  pentru numărul de noduri ale grafului,  $k$  pentru nodul care se prelucurează,  $i$  și  $j$  pentru parcurgerea matricei de adiacență și a vectorilor.
2. **Matricea de adiacență a grafului  $a$ .**
3. **Vectorul de vizitare vizitat.** El conține  $n$  elemente în care se înregistrează nodurile vizitate. Elementele vectorului  $vizitat[i]$  sunt definite astfel:

$$vizitat[i] = \begin{cases} 1, & \text{dacă nodul } i \text{ a fost vizitat} \\ 0, & \text{dacă nodul } i \text{ nu a fost vizitat încă} \end{cases}$$

4. **Coadă de așteptare  $c$  a nodurilor parcurse.** Coadă de așteptare  $c$  gestionează nodurile prelucrate. Pentru coada de așteptare  $c$ , se folosește implementarea statică cu un vector  $c$  cu  $n$  elemente. Pentru gestionarea cozii de așteptare, se folosesc două variabile de memorie **prim** și **ultim** care joacă rolul de indicatori pentru a memora adresa primului element din listă (capul cozii) și, respectiv, adresa ultimului element din listă (coada cozii). În această coadă de așteptare vor fi înregistrate nodurile vizitate, în ordinea în care au fost vizitate. Inițial, coada de așteptare  $c$  conține un singur element care corespunde nodului



cu care începe parcurgerea grafului, iar valoarea indicatorilor **prim** și **ultim** este 1. Pe măsură ce se parcurge graful, se completează următoarele elemente ale vectorului **c**. Atunci când se prelucrează un nod **k** de la capul cozii de așteptare, prin coada cozii de așteptare se introduc toate nodurile **i** vecine cu nodul **k** care nu au fost vizitate încă.

**Algoritmul** pentru parcurgerea grafului este următorul:

- PAS1.** Se citește din fișier, valoarea pentru variabila **n** și matricea de adiacență a grafului.
- PAS2.** Se inițializează cu 0 elementele vectorului de vizitare **vizitat**, deoarece inițial nici unul dintre noduri nu a fost vizitat.
- PAS3.** Se introduce de la tastatură valoarea variabilei de memorie **k**, corespunzătoare primului nod cu care începe parcurgerea grafului.
- PAS4.** Se inițializează coada de așteptare **c** (indicatorii și conținutul primului element introdus în coada de așteptare), astfel: **prim**←1, **ultim**←1 și **c[prim]**←**k**, deoarece, în coada de așteptare **c** este înregistrat doar nodul **k** cu care începe parcurgerea.
- PAS5.** Se actualizează vectorul **vizitat** – atribuind elementului **k** al vectorului valoarea 1, deoarece a fost vizitat (**vizitat[k]**←1).
- PAS6.** **Cât timp** coada nu este vidă (**prim** ≤ **ultim**) **execută**
- PAS7.** Se extrage următorul element din coada de așteptare, corespunzător nodului cărui **i** se vor vizita vecinii (**k**←**c[prim]**);).
- PAS8.** Pentru toate nodurile **i** vecine ale vârfului **k**, nevizitate încă, **execută**
- PAS9.** Se incrementează valoarea indicatorului **ultim**, pentru a înregistra următorul nod care va trebui vizitat, prin adăugare la coada cozii de așteptare (**ultim**←**ultim**+1);).
- PAS10.** Se adaugă, la sfârșitul cozii de așteptare **c**, nodul **i** vecin nodului **k**, care nu a fost încă vizitat (**c[ultim]** ←**i**);).
- PAS11.** Prin adăugarea nodului **j** la coada de așteptare **c**, se consideră că acest nod a fost vizitat – și se actualizează elementul din vectorul de vizitare care corespunde acestui nod (**vizitat[i]**←1);). Se trece la Pasul 6.
- PAS12.** Se pregătește indicatorul **prim** pentru a se extrage următorul nod, din coada de așteptare, ai cărui vecini vor fi vizitați (**prim**←**prim**+1);). Se trece la Pasul 7.
- PAS13.** Se afișează lista nodurilor vizitate, în ordinea în care au fost vizitate, prin extragerea lor din vectorul **c**, pornind de la elementul 1 și până la elementul **n**.

Pentru graful  $G_{26}$ , pornind din nodul 1, nodurile în coada de așteptare sunt adăugate astfel:

|         |   |   |   |   |   |   |   |   |   |    |
|---------|---|---|---|---|---|---|---|---|---|----|
| vizitat | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |    |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

|          |   |  |  |  |  |  |  |  |  |  |
|----------|---|--|--|--|--|--|--|--|--|--|
| <b>c</b> | 1 |  |  |  |  |  |  |  |  |  |
|----------|---|--|--|--|--|--|--|--|--|--|

**Inițializarea** cozii de așteptare cu primul nod: **prim**=1; **ultim**=1.

|         |   |   |   |   |   |   |   |   |   |    |
|---------|---|---|---|---|---|---|---|---|---|----|
| vizitat | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |    |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

|          |   |   |   |   |  |  |  |  |  |  |
|----------|---|---|---|---|--|--|--|--|--|--|
| <b>c</b> | 1 | 2 | 3 | 4 |  |  |  |  |  |  |
|----------|---|---|---|---|--|--|--|--|--|--|

**Prelucrarea** nodului: **prim**=1 ⇒ **ultim**=4.

|         |   |   |   |   |   |   |   |   |   |    |
|---------|---|---|---|---|---|---|---|---|---|----|
| vizitat | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |    |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

|          |   |   |   |   |   |  |  |  |  |  |
|----------|---|---|---|---|---|--|--|--|--|--|
| <b>c</b> | 1 | 2 | 3 | 4 | 5 |  |  |  |  |  |
|----------|---|---|---|---|---|--|--|--|--|--|

**Prelucrarea** nodului: **prim**=2 ⇒ **ultim**=5.

|                                                               |   |   |   |   |   |   |   |   |   |    |
|---------------------------------------------------------------|---|---|---|---|---|---|---|---|---|----|
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |    |
| Prelucrarea nodului: prim=3 ⇒ ultim=7.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |    |
| Prelucrarea nodului: prim=4 ⇒ ultim=7.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |    |
| Prelucrarea nodului: prim=5 ⇒ ultim=8.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |    |
| Prelucrarea nodului: prim=6 ⇒ ultim=9.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |    |
| Prelucrarea nodului: prim=7 ⇒ ultim=9.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |    |
| Prelucrarea nodului: prim=8 ⇒ ultim=9.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |    |
| Prelucrarea nodului: prim=9 ⇒ ultim=9.                        |   |   |   |   |   |   |   |   |   |    |
| vizitat                                                       | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |
|                                                               | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| c                                                             | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |    |
| Prelucrarea nodului: prim=10 ⇒ ultim=9. Terminare: prim>ultim |   |   |   |   |   |   |   |   |   |    |

Pentru implementarea algoritmului, se folosesc subprogramele:

- funcția procedurală **citeste** creează matricea de adiacență, prin preluarea informațiilor din fișierul **f** (dimensiunea matricei de adiacență și matricea);
- funcția procedurală **init** inițializează coada de așteptare cu primul nod vizitat;
- funcția operand **este\_vida** testează coada de așteptare dacă este vidă;
- funcția procedurală **adauga** adaugă un nod la coada de așteptare;
- funcția procedurală **elimina** elimină nodul prelucrat din coada de așteptare;
- funcția procedurală **prelucrare** prelucerează primul nod din coadă: adaugă la coada de așteptare toți vecinii nevizați ai acestui nod și apoi îl elimină din coada de așteptare;
- funcția procedurală **afisare** afișează nodurile grafului în ordinea prelucrării.

Pentru testarea programului, se folosește graful neorientat  $G_{22}$ , a cărui matrice de adiacență se citește din fișierul text *graf21.txt*.

```
int n,a[10][10],vizitat[20],c[20],prim,ultim,k;
fstream f("graf21.txt",ios::in);
void citeste()
{ //funcția prin care se citește matricea de adiacență din fișier}
void init (int k) {prim=ultim=1; c[ultim]=k; vizitat[k]=1;}
int este vida() {return ultim<prim;}
void adauga(int i) {ultim++; c[ultim]=i; vizitat[i]=1;}
void elimina() {prim++;}
void prelucrare()
{int i; k=c[prim];
 for (i=1;i<=n;i++) if (a[k][i]==1 && vizitat[i]==0) adauga(i);
 elimina();}
void afisare()
{for (int i=1;i<=n;i++) cout<<c[i]<<" ";}
void main()
{citeste(); cout<<"nod de pornire: "; cin>>k; init(k);
 while (!este vida()) prelucrare();
 cout<<"Nodurile vizitate prin metoda BF sunt: "<<endl;
 afisare();}
```

### Atenție

Prin parcurgerea în lățime a unui graf, determinați lanțurile, respectiv drumurile de lungime minimă.

## 2.6.8.2. Parcurgerea în adâncime – Depth First

Pentru această metodă de parcurgere a unui graf cu  $n$  noduri se folosesc următoarele structuri de date și date elementare:

1. **Variabilele de memorie.**  $n$  pentru numărul de noduri ale grafului,  $k$  pentru nodul care se prelucrează,  $i$  și  $j$  pentru parcurgerea matricei de adiacență și a vectorilor.
2. **Matricea de adiacență a grafului  $a$ .**
3. **Vectorul de vizitare  $vizitat$ .** El conține  $n$  elemente în care se înregistrează nodurile vizitate. Elementele vectorului  $vizitat[i]$  sunt definite ca și în metoda precedentă de parcurgere.
4. **Stiva  $st$  a nodurilor parcurse.** Stiva  $st$  gestionează nodurile vecine parcurse în adâncime. Pentru gestionarea stivei se folosește variabila de memorie  $vf$  pentru a identifica ultimul nod intrat în stivă. Pentru stiva  $st$  se folosește implementarea statică cu un vector  $st$  cu  $n$  elemente. În această stivă vor fi înregistrate în ordine nodurile vizitate în adâncime, pe o direcție. Inițial stiva  $st$  conține un singur element care corespunde nodului cu care începe parcurgerea grafului, iar valoarea variabilei  $vf$  este 1. Pe măsură ce se parcurge graful se completează următoarele elemente ale vectorului  $st$ . Atunci când se prelucrează un nod  $k$ , pe la vârful stivei se introduc în stivă toate nodurile  $i$  vecine cu nodul  $k$  care nu au fost vizitate încă.

Algoritmul pentru parcurgerea grafului este următorul:

**PAS1.** Se citesc din fișier valoarea pentru  $n$  și matricea de adiacență a grafului.

**PAS2.** Se inițializează cu 0 elementele vectorului de vizitare  $vizitat$ , deoarece nici unul dintre noduri nu a fost vizitat.

- PAS3.** Se introduce de la tastatură valoarea variabilei de memorie  $k$ , corespunzătoare primului nod cu care începe parcurgerea grafului și se afișează eticheta lui.
- PAS4.** Se inițializează stiva  $st$  (vârful și conținutul vârfului stivei), astfel:  $vf \leftarrow 1$ ;  $st[vf] \leftarrow k$ ; deoarece inițial în stiva  $st$  este înregistrat doar nodul  $k$  cu care începe parcurgerea.
- PAS5.** Se actualizează vectorul  $vizitat$  atribuind elementului  $k$  al vectorului valoarea 1 deoarece a fost vizitat ( $vizitat[k] \leftarrow 1$ ).
- PAS6.** Cât timp stiva nu este vidă ( $vf > 0$ ) execută
- PAS7.** Se extrage din vârful stivei, elementul corespunzător nodului căruia  $i$  se vor vizita vecinii ( $k \leftarrow st[vf]$ );).
- PAS8.** Se inițializează nodul  $i$  cu care începe căutarea ( $i \leftarrow 1$ );).
- PAS9.** Cât timp nu s-a găsit un nod  $i$  vecin nodului  $k$ , nevizitat încă ( $i \leq n$  și ( $a[i][k] = 0$  sau ( $a[i][k] = 1$  și  $vizitat[i] = 1$ ))) execută
- PAS10.** Se trece la următorul nod în vederea verificării ( $i \leftarrow i + 1$ ); și se revine la Pasul 9.
- PAS11.** Dacă nu s-a mai găsit un nod  $i$  vecin nodului  $k$  nevizitat încă, atunci se elimină nodul  $k$  din stivă prin coborârea vârfului stivei ( $vf \leftarrow vf - 1$ ), altfel se afișează nodul găsit  $i$ , se adaugă în vârful stivei ( $vf \leftarrow vf + 1$ ;  $st[vf] \leftarrow i$ ;) și se actualizează elementul din vectorul de vizitare care corespunde acestui nod deoarece prin adăugarea nodului  $i$  la stiva  $st$  se consideră că acest nod a fost vizitat ( $vizitat[i] \leftarrow 1$ ;) și se trece la Pasul 6.

Pentru graful  $G_{26}$ , pornind din nodul 1, nodurile în stivă sunt adăugate astfel:

|         |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|
| vizitat | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| st      | 1 |   |   |   |   |   |   |   |   |

Inițializarea stivei cu primul nod:  $vf = 1$ .

|         |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|
| vizitat | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| st      | 1 | 2 |   |   |   |   |   |   |   |

Prelucrarea nodului:  $vf = 1 \Rightarrow vf = 2$ .

|         |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|
| vizitat | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| st      | 1 | 2 | 5 |   |   |   |   |   |   |

Prelucrarea nodului:  $vf = 2 \Rightarrow vf = 3$ .

|         |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|
| vizitat | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| st      | 1 | 2 | 5 | 3 |   |   |   |   |   |

Prelucrarea nodului:  $vf = 3 \Rightarrow vf = 4$ .

|         |   |   |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|---|---|
| vizitat | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| st      | 1 | 2 | 5 | 3 | 6 |   |   |   |   |

Prelucrarea nodului:  $vf = 4 \Rightarrow vf = 5$ .

**Prelucrarea nodului:**  $vf=5 \Rightarrow vf=6$ .

**Prelucrarea nodului:  $vf=6 \Rightarrow vf=5$ .**

**Prelucrarea nodului:  $vf=5 \Rightarrow vf=6$ .**

**Prelucrarea nodului:**  $vf=6 \Rightarrow vf=5$ .

**Prelucrarea nodului:  $vf=5 \Rightarrow vf=4$ .**

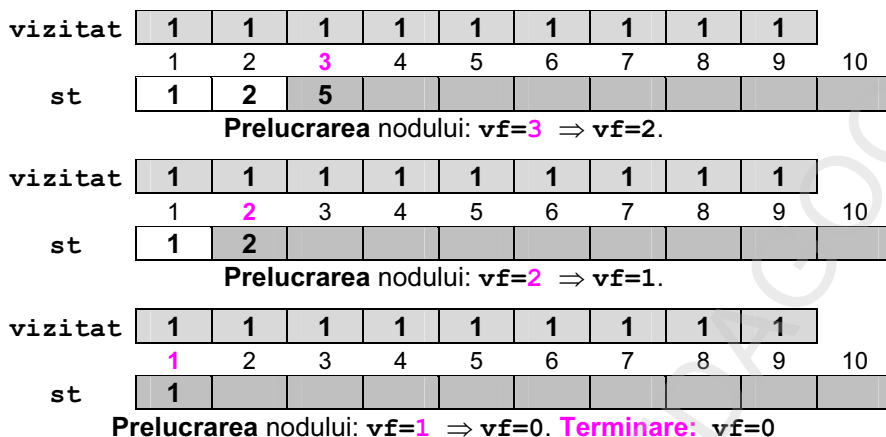
**Prelucrarea nodului:**  $vf=4 \Rightarrow vf=5$ .

**Prelucrarea nodului:  $vf=5 \Rightarrow vf=6$ .**

**Prelucrarea nodului:  $vf=6 \Rightarrow vf=5$ .**

**Prelucrarea nodului:  $vf=5 \Rightarrow vf=4$ .**

**Prelucrarea nodului:**  $vf=4 \Rightarrow vf=3$ .



Pentru implementarea algoritmului se folosesc subprogramele:

- funcția procedurală **citește** creează matricea de adiacență prin preluarea informațiilor din fișierul text **f**;
- funcția procedurală **init** inițializează stiva cu primul nod vizitat;
- funcția operand **este\_vida** testează stiva dacă este vidă;
- funcția procedurală **adauga** adaugă un nod la stivă;
- funcția procedurală **elimina** elimină nodul din vârful stivei;
- funcția procedurală **prelucrare** prelucerează nodul din vârful stivei: caută primul vecin nevizitat; dacă găsește un astfel de nod îl afișează și îl adaugă în stivă, altfel nodul din vârful stivei este eliminat (nu mai are vecini nevizitați);

Pentru testarea programului se folosește graful neorientat  $G_{22}$  a cărui matrice de adiacență se citește din fișierul text *graf21.txt*.

**Observație.** Pentru grafurile orientate, în funcția **prelucrare()** condiția structurii repetitive **while** este:  $i \leq n \ \&\& \ ((a[k][i]==0 \ || \ a[i][k]==0) \ || \ ((a[k][i]==1 \ || \ a[i][k]==1) \ \&\& \ vizitat[i]==1))$

```
int n,a[10][10],vizitat[20],st[20],vf,k;
fstream f("graf21.txt",ios::in);
void citeste()
{//funcția prin care se citește matricea de adiacență din fișier}
void init (int k) {vf=1; st[vf]=k; vizitat[k]=1;}
int este_vida() {return vf==0;}
void adauga(int i) {vf++; st[vf]=i; vizitat[i]=1;}
void elimina() {vf--;}
void prelucrare()
{int i=1; k=st[vf];
 while (i<=n && (a[k][i]==0 || (a[k][i]==1 && vizitat[i]==1))) i++;
 if (i==n+1) elimina();
 else {cout<<i<<" "; adauga(i);}}
void main()
{citeste();
cout<<"nodul de pornire: "; cin>>k;
cout<<"Nodurile vizitate prin metoda DF sunt: "<<endl;
cout<<k<<" "; init(k);
while (!este_vida()) prelucrare();}
```

**Atenție**

Prin **parcurerea în adâncime** a unui graf determinați **nodurile care aparțin aceleiași componente conexe, respectiv subgrafului de succesori ai unui nod.**

**Temă**

Scrieți un program care citește din fișierul text *graf4.txt* lista arcelor unui graf orientat și care, folosind algoritmul corespunzător de parcurgere a unui graf orientat, realizează:

- Afișează lanțurile de lungime minimă dintre noduri.
- Verifică dacă două noduri  $x$  și  $y$  aparțin aceleiași componente conexe (și varianta, aparțin aceleiași componente tare conexe). Etichetele nodurilor se citesc de la tastatură.
- Verifică dacă graful este tare conex și afișează un mesaj de informare.

## 2.7. Arborele

### 2.7.1. Definiția arborelui

Se numește **arbore**  $A$  un **graf neorientat conex și fără cicluri**.

Se numește **subarbore** al arborelui  $A=(X,U)$  orice arbore  $S=(Y,V)$  care are proprietatea:  $Y \subseteq X$  și  $V \subseteq U$ .

**Exemplu.** Graful  $G_{23}$  cu 8 noduri din figura 24 este un arbore (un graf neorientat conex și aciclic), iar graful  $G_{24}$  este un subarbore al acestuia.

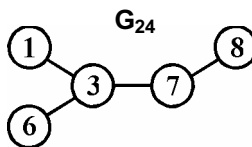
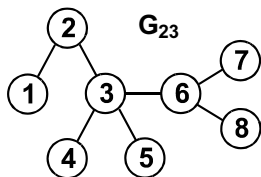


Fig. 24

#### Teorema 17

Următoarele definiții sunt echivalente – pentru un graf  $G$  cu  $n$  noduri și  $m$  muchii:

- $G$  este un **arbore**.
- $G$  este un **graf aciclic** cu  $n-1$  muchii.
- $G$  este un **graf conex** cu  $n-1$  muchii.
- $G$  este un **graf fără cicluri maximal** (dacă în graful fără cicluri  $G$  unim două noduri oarecare neadiacente printr-o muchie, graful obținut conține un ciclu).
- $G$  este un **graf conex minimal** (dacă în graful conex  $G$  suprimăm o muchie oarecare, graful obținut nu mai este conex).
- Orice pereche de noduri este legată printr-un lanț și numai unul.**

**Demonstrație.** Este suficient să se demonstreze implicațiile  $(1) \Rightarrow (2)$ ,  $(2) \Rightarrow (3)$ ,  $(3) \Rightarrow (4)$ ,  $(4) \Rightarrow (5)$ ,  $(5) \Rightarrow (6)$  și  $(6) \Rightarrow (1)$ . Prin tranzitivitatea acestor implicații rezultă implicațiile  $(2) \Rightarrow (1)$ ,  $(3) \Rightarrow (2)$ ,  $(4) \Rightarrow (3)$ ,  $(5) \Rightarrow (4)$ ,  $(6) \Rightarrow (5)$  și  $(1) \Rightarrow (6)$ . Din aceste implicații rezultă că  $(1) \Leftrightarrow (2)$ ,  $(2) \Leftrightarrow (3)$ ,  $(3) \Leftrightarrow (4)$ ,  $(4) \Leftrightarrow (5)$ ,  $(5) \Leftrightarrow (6)$  și  $(6) \Leftrightarrow (1)$ . Din tranzitivitatea relației de echivalență rezultă că oricare două din cele 6 propoziții sunt echivalente.

**(1)  $\Rightarrow$  (2).** Ipoteza: Graful  $G$  este conex și aciclic – din definiția arborelui (1).

**Concluzie:** Graful  $G$  este aciclic și are  $n-1$  muchii (2).

Proprietatea că graful  $G$  este aciclic este comună ipotezei și concluziei. Trebuie demonstrat doar că un graf conex aciclic are  $n-1$  muchii. Dacă  $G$  este conex aciclic, nu trebuie eliminată nici o muchie pentru a se obține un graf parțial conex aciclic. Cum numărul de muchii care trebuie eliminate dintr-un



graf conex pentru a obține un graf parțial conex aciclic este egal cu  $m-n+1$ , înseamnă că în acest caz  $m-n+1=0$ . Rezultă că  $m=n-1$ .

(2) $\Rightarrow$ (3). *Ipoteza:* Graful  $G$  este aciclic și are  $n-1$  muchii – din definiția arborelui (2).

*Concluzie:* Graful  $G$  este conex și are  $n-1$  muchii (3).

Proprietatea că graful  $G$  are  $n-1$  muchii este comună ipotezei și concluziei. Trebuie demonstrat doar că un graf cu  $n-1$  muchii fiind aciclic este și conex. Se știe că, într-un graf cu  $p$  componente conexe, numărul de muchii care trebuie eliminate pentru a obține un graf parțial aciclic este egal cu  $m-n+p$ . Graful  $G$  este aciclic ( $m-n+p=0$ ) și are  $n-1$  muchii ( $m=n-1$ ) și  $(n-1)-n+p=0$ . Rezultă că  $p=1$  (graful are o singură componentă conexă, deci este conex).

(3) $\Rightarrow$ (4). *Ipoteza:* Graful  $G$  este conex și are  $n-1$  muchii (3).

*Concluzie:* Graful  $G$  este aciclic maximal (4).

$G$  fiind conex, numărul de componente conexe  $p$  este egal cu 1. Numărul de muchii  $m$  ale grafului  $G$  este egal cu  $n-1$ . Rezultă că numărul de muchii care trebuie eliminate din graful  $G$  ca să se obțină un graf parțial aciclic este egal cu:  $m-n+p = (n-1)+n+1 = 0$ , adică nici o muchie. Rezultă că graful  $G$  este aciclic. Graful este maximal pentru această proprietate, deoarece fiind conex, orice muchie  $[x_i, x_j]$  care se va adăuga va forma un ciclu cu lanțul  $L(x_i, x_j)$  – existența acestui lanț rezultă din conexitatea grafului  $G$ .

(4) $\Rightarrow$ (5). *Ipoteza:* Graful  $G$  este aciclic maximal (4).

*Concluzie:* Graful  $G$  este conex minimal (5).

a. Presupunem că graful  $G$  nu este conex. El are cel puțin două componente conexe:  $C_1$  și  $C_2$ . Înseamnă că există două noduri  $x \in C_1$  și  $y \in C_2$  pe care le putem lega cu muchia  $[x, y]$ . Dar, din ipoteză, rezultă că orice muchie am adăuga la graf, el va conține un ciclu. Rezultă că prin adăugarea muchiei  $[x, y]$  graful va conține un ciclu și că, între nodurile  $x$  și  $y$  există deja un lanț și ele aparțin aceleiași componente conexe, ceea ce contrazice presupunerea făcută. Înseamnă că graful  $G$  este conex.

b. Presupunem că graful  $G$  care este conex nu este minimal cu această proprietate. Înseamnă că prin eliminarea unei muchii se obține graful parțial  $H$  care are  $n$  noduri și  $m-1$  muchii, este conex și aciclic. Graful  $G$  fiind conex și aciclic, înseamnă că numărul de muchii care trebuie eliminate pentru a obține un graf parțial conex aciclic este egal cu 0, adică  $m-n+1=0$  și  $m=n-1$ . Numărul de muchii ale grafului  $H$  este egal cu  $m-1=n-2$ . Fiind aciclic înseamnă că numărul de muchii care trebuie eliminate pentru a obține un graf parțial aciclic este egal cu 0, adică  $(n-2)-n+p=0$ . Rezultă că  $p=2$ , adică graful parțial  $H$  conține două componente conexe, ceea ce contrazice presupunerea că el este conex.

(5) $\Rightarrow$ (6). *Ipoteza:* Graful  $G$  este conex minimal (5).

*Concluzie:* Orice pereche de noduri este legată printr-un lanț și numai unul (6).

Graful  $G$  fiind conex, înseamnă că există cel puțin un lanț care leagă oricare două noduri  $x$  și  $y$ . Presupunem că există cel puțin două lanțuri între nodurile  $x$  și  $y$ :  $L_1$  și  $L_2$ . Înseamnă că, suprimând o muchie din lanțul al doilea, graful rămâne conex, deoarece nodurile  $x$  și  $y$  vor fi legate prin lanțul  $L_1$ , ceea ce contrazice ipoteza că graful  $G$  este conex minimal. Rezultă că cele două noduri  $x$  și  $y$  nu sunt legate decât printr-un singur lanț.

(6) $\Rightarrow$ (5). *Ipoteza:* Orice pereche de noduri este legată printr-un lanț și numai unul (6).

*Concluzie:* Graful  $G$  este conex și aciclic – din definiția arborelui (1).

Deoarece în graful  $G$  orice pereche de noduri  $x$  și  $y$  este legată printr-un lanț, înseamnă că graful  $G$  este conex. Presupunem că graful  $G$  conține cel puțin un ciclu. Considerând două noduri oarecare  $x$  și  $y$  care aparțin acestui ciclu, înseamnă că între cele două noduri există două lanțuri diferite, ceea ce contrazice ipoteza. Rezultă că graful  $G$  este aciclic.

## Propoziția 9

**Orice arbore cu  $n$  noduri are  $n-1$  muchii.**

**Demonstrație.** Arborele fiind un graf conex minimal, înseamnă că are  $n-1$  muchii.

**Propoziția 10.** Orice arbore cu  $n \geq 2$  noduri conține cel puțin două noduri terminale.

**Demonstrație** – prin reducere la absurd. Presupunem că nu există decât un singur nod terminal  $x$ . Oricare alt nod din arbore are cel puțin gradul 2. Alegem din arbore, dintre lanțurile elementare care au o extremitate în nodul  $x$ , lanțul de lungime maximă:  $L(x,y)$ . Nodul  $y$  având gradul mai mare decât 1, înseamnă că mai există, în afara nodului care îl precede în lanț, cel puțin un nod  $z$  care este adiacent cu el. Lanțul fiind elementar, înseamnă că nodul  $y$  nu există în lanț decât la extremitatea lui, iar muchia  $[y,z]$  nu face parte din lanț și putem adăuga această muchie la lanț. Lanțul fiind de lungime maximă, înseamnă că nodul  $z$  aparține lanțului și prin adăugarea acestei muchii la lanț se închide un ciclu, ceea ce contrazice definiția arborelui (graf aciclic).

### Temă



1. Precizați dacă este arbore graful  $G_{25} = (X_{40}, U_{40})$  definit astfel:

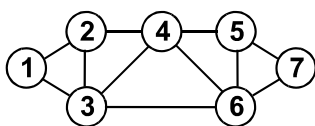
$$X_{25} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

$$U_{25} = \{[1,2], [1,3], [1,5], [2,6], [2,7], [3,4], [3,9], [4,8], [8,10]\}.$$

2. Scrieți un program care citește dintr-un fișier text informații despre un graf neorientat (de pe primul rând, numărul de noduri ale grafului –  $n$ , iar de pe următoarele  $n$  rânduri matricea de adiacență a grafului) și care:
- verifică dacă graful este un arbore;
  - dacă nu este arbore, verifică dacă prin eliminarea muchiilor poate fi făcut arbore; în caz afirmativ să se specifice câte muchii trebuie eliminate și care sunt aceste muchii.
  - dacă nu este arbore, verifică dacă prin adăugarea muchiilor poate fi făcut arbore; în caz afirmativ să se specifice câte muchii trebuie adăugate și care sunt aceste muchii.

## 2.7.2. Arborele parțial

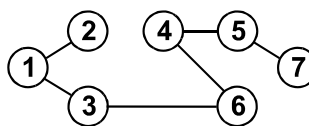
Dacă un **graf parțial** al unui graf  $G$  este arbore el se numește **arbore parțial** al grafului  $G$ .



Graful  $G_{26}$



Fig. 25



Arbore parțial al grafului  $G_{26}$

### Teorema 18

Un graf  $G$  conține un **arbore parțial** dacă și numai dacă este un **graf conex**.

**Demonstrație.** Notăm cele două propoziții, astfel:

- (1) – Graful  $G$  conține un arbore parțial.
- (2) – Graful  $G$  este conex.

Trebuie să demonstrăm că  $(1) \Rightarrow (2)$  și  $(2) \Rightarrow (1)$ .

**(1)  $\Rightarrow$  (2).** Să considerăm că graful  $G$  conține arborele parțial  $H$ . Din definiția arborelui rezultă că  $H$  este un graf conex. Din definiția arborelui parțial rezultă că  $H$  este graf parțial al grafului  $G$ . Deoarece graful  $G$  se obține prin adăugarea de muchii la un graf conex ( $H$ ), este și el un graf conex.

**(2)  $\Rightarrow$  (1).** Dacă  $G$  este un graf conex minimal, înseamnă că este un arbore (din Teorema 20:  $(1) \Leftrightarrow (5)$ ) și  $H=G$ . Dacă  $G$  nu este un graf conex minimal, înseamnă că există o muchie  $[x,y]$  pe care o putem elimina astfel încât să obținem un graf parțial conex  $G_1$ . Dacă graful parțial  $G_1$  este un graf conex minimal, înseamnă că  $H = G_1$ ; altfel se repetă procesul de eliminare a câte unei muchii până se obține un graf parțial conex minimal. Acesta va fi arborele parțial  $H$ .



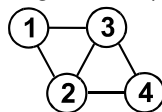
10. Numărul de subgrafuri cu 3 noduri care se pot obține din graf este:  
 a. 10                      b. 3                      c. 20                      d. 56
11. Numărul de grafuri parțiale cu 7 muchii care se pot obține din graf este:  
 a. 72                      b. 18                      c. 36                      d. 9
12. Numărul minim de muchii care se pot elimina din componenta conexă  $\{1,2,3,4,5,7\}$  a grafului pentru a se obține un graf parțial neconex este:  
 a. 1                      b. 4                      c. 3                      d. 2
13. Numărul maxim de muchii care se pot elimina din componenta conexă  $\{1,2,3,4,5,7\}$  a grafului pentru a se obține un graf parțial conex este:  
 a. 5                      b. 4                      c. 3                      d. 2

Următorii 9 itemi se referă la graful orientat cu 6 noduri și având arcele  $[1,2]$ ,  $[1,3]$ ,  $[1,4]$ ,  $[1,5]$ ,  $[2,3]$ ,  $[3,1]$ ,  $[3,4]$ ,  $[3,6]$ ,  $[4,1]$ ,  $[4,5]$ ,  $[4,6]$ ,  $[6,4]$ .

14. Numărul de noduri care au gradul intern egal cu gradul extern este:  
 a. 1                      b. 2                      c. 4                      d. 3
15. Numărul de elemente de 0 din matricea de adiacență este:  
 a. 9                      b. 18                      c. 26                      d. 24
16. Suma gradelor externe și a gradelor interne ale tuturor nodurilor este:  
 a. 12                      b. 24                      c. 20                      d. 14
17. Numărul de cicluri elementare de lungime 4 este:  
 a. 4                      b. 3                      c. 0                      d. 6
18. Numărul de lanțuri elementare de lungime 4 este:  
 a. 16                      b. 8                      c. 10                      d. 22
19. Numărul de circuite elementare de lungime 4 este:  
 a. 2                      b. 4                      c. 3                      d. 5
20. Numărul de lanțuri elementare între nodul 1 și nodul 6 este:  
 a. 6                      b. 3                      c. 4                      d. 8
21. Numărul de drumuri elementare între nodul 1 și nodul 6 este:  
 a. 6                      b. 3                      c. 4                      d. 5
22. Numărul de componente tare conexe este:  
 a. 1                      b. 3                      c. 2                      d. 4
23. Se consideră graful neorientat dat prin matricea de adiacență alăturată. În graf, nodurile sunt numerotate de la 1 la 4, corespunzător liniilor tabloului. Să se determine câte perechi de noduri neadiacente există în graf. Se consideră perechile neordonate (perechea 1–2 este aceeași cu perechea 2–1).  
 a. 1                      b. 2                      c. 3                      d. 4

(Bacalaureat – Sesiunea august 2003)

24. Se consideră graful neorientat din figura alăturată. Să se determine care dintre următoarele liste reprezintă lista de adiacență a nodului 3:



- a. 1 2 4                      b. 3 1 2 4                      c. 2 4                      d. 1 3 4

(Bacalaureat – Sesiunea iunie-iulie 2003)

25. Se consideră un graf orientat având matricea de adiacență alăturată. Stabiliți care dintre nodurile grafului au gradul intern egal cu gradul extern.

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |

- a. 3                      b. 1                      c. 0                      d. 2

(Bacalaureat – Simulare 2003)

26. Într-un graf neorientat cu  $n$  noduri numărul maxim de muchii ce pot exista este:

- a.  $n \times (n+1)/2$                       b.  $n \times (n-1)/2$                       c.  $n \times (n-1)$                       d.  $n^2$

(Bacalaureat – Sesiunea iunie-iulie 2003)

27. Numărul maxim de muchii într-un graf neorientat cu 8 noduri care nu conține niciun ciclu este:

- a. 4                                      b. 8                                      c. 27                                      d. 7  
(Bacalaureat – Simulare 2005)

28. Știind că un graf neorientat fără noduri izolate are 7 muchii și 8 noduri, stabiliți numărul maxim de componente conexe din care poate fi format acesta.

- a. 1                                      b. 2                                      c. 3                                      d. 4  
(Bacalaureat – Sesiunea august 2005)

29. Numărul minim de noduri dintr-un graf neorientat cu 12 muchii, fără noduri izolate, graf format din exact 3 componente conexe este:

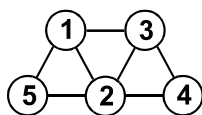
- a. 7                                      b. 8                                      c. 9                                      d. 10  
(Bacalaureat – Sesiunea specială 2005)

30. Numărul minim de componente conexe ale unui graf neorientat cu 9 muchii și 12 noduri este:

- a. 2                                      b. 3                                      c. 4                                      d. 5  
(Bacalaureat – Sesiunea iunie-iulie 2005)

31. Se consideră graful neorientat din figura alăturată. Să se determine eticheta nodului care are lista de adiacență 2 3 5.

- a. 5                                      b. 4                                      c. 3                                      d. 1



(Bacalaureat – Sesiunea iunie-iulie 2003)

32. Se consideră un graf orientat dat prin listele de adiacență: 1) 2 4 2) 4 3) 1 5 4) 4 2 3 5) 2 3 4. Stabiliți numărul de noduri care au gradul intern egal cu gradul extern.

- a. 3                                      b. 1                                      c. 3                                      d. 2  
(Bacalaureat – Sesiunea iunie-iulie 2003)

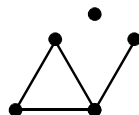
33. Care este numărul minim de muchii care pot fi plasate într-un graf neorientat cu 53 de noduri astfel încât să nu existe nici un nod izolat.

- a. 27                                      b. 26                                      c. 53                                      d. 52  
(Bacalaureat – Sesiunea iunie-iulie 2003)

34. Se consideră un graf neorientat G cu 8 noduri și 16 muchii. Numărul de noduri izolate din G este:

- a. exact 1                                      b. exact 0                                      c. cel mult 1                                      d. cel mult 2  
(Bacalaureat – Sesiunea iunie-iulie 2003)

35. Stabiliți care dintre următoarele matrice de adiacență corespunde grafului din figura alăturată.



- a) 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |

      b) 

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |

      c) 

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

      d) 

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

(Bacalaureat – Sesiunea august 2003)

36. Care este matricea de adiacență a unui graf neorientat cu 4 noduri, 2 muchii și cel puțin un nod izolat?

- a) 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |

      b) 

|   |   |   |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

      c) 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |

      d) 

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |

(Bacalaureat – Simulare 2004)

37. Într-un graf neorientat cu 20 de noduri numerotate de la 1 la 20 există muchii numai între perechile de noduri  $i$  și  $j$  cu proprietatea  $\text{abs}(i-j) > 1$ . Numărul de valori egale cu 1 din matricea de adiacență corespunzătoare grafului este:

- a. 190      b. 362      c. 340      d. 171

(Bacalaureat – Sesiunea iunie-iulie 2003)

38. Stabiliți care dintre următoarele variante este matricea de adiacență a unui subgraf al grafului din figura alăturată:



a)

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

b)

|   |   |   |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |

c)

|   |   |   |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 0 | 1 |

d)

|   |   |   |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(Bacalaureat – Sesiunea specială 2003)

39. Se consideră un graf orientat cu nodurile numerotate cu 1, 2, 3, 4, 5, corespunzător liniilor matricei de adiacență alăturate. Stabiliți dacă nodurile 1 și 3:

- a. aparțin aceleiași componente conexe      b. sunt noduri izolate  
c. sunt conectate printr-un lanț      d. sunt noduri adiacente

(Bacalaureat – Simulare 2003)

40. Se consideră graf neorientat dat prin matricea de adiacență alăturată. În graf, nodurile sunt numerotate de la 1 la 4, corespunzător liniilor tabloului. Să se determine lungimea maximă a unui lanț ce unește nodurile 1 și 3:

- a. 1      b. 2      c. 3      d. 4

(Bacalaureat – Sesiunea iunie-iulie 2003)

41. Care dintre următoarele secvențe reprezintă un drum în graf orientat dat prin matricea de adiacență alăturată, știind că nodurile sunt numerotate de la 1 la 4 corespunzător liniilor și coloanelor tabloului.

- a. 3 4 3 2 4      b. 3 1 2 3      c. 2 1 3 4      d. 4 3 1 2

(Bacalaureat – Sesiunea specială 2004)

42. Se consideră un graf orientat cu 6 noduri etichetate cu numere de la 1 la 6 și 6 arce astfel încât există un arc de la fiecare nod cu eticheta  $i$  către un nod cu eticheta  $i \times 2$  dacă există un astfel de nod sau către nodul cu eticheta  $i-1$  în caz contrar. Care este lungimea maximă a unui drum în graf?

- a. 4      b. 3      c.  $\infty$       d. 2

(Simulare Bacalaureat 2003)

43. Se consideră un graf neorientat cu 7 noduri și 3 muchii. Numărul de componente conexe din care poate fi format grafurile este:

- a. exact 4      b. 4 sau 5      c. 3 sau 4      d. cel puțin 5

(Bacalaureat – Simulare 2003)

44. Dacă într-un graf neorientat conex cu  $n$  noduri, lista de adiacență a fiecărui nod este formată din exact două elemente, atunci în grafurile respective există:

- a. cel puțin  $n/2$  cicluri      b. exact  $n/2$  cicluri  
c. exact un ciclu      d. cel puțin două cicluri

(Bacalaureat – Sesiunea specială 2003)

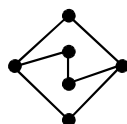
45. Numărul minim de muchii ce se pot alege pentru a fi eliminate din grafurile neorientate din figura alăturată astfel încât acestea să devină neconexe este:

- a. 2      b. 4      c. 3      d. 1

(Bacalaureat – Sesiunea iunie-iulie 2003)

46. Numărul de grafuri conexe aciclice care se pot obține cu 4 noduri este:

- a. 2      b. 4      c. 28      d. 24



47. Se consideră un graf orientat tare conex. Stabiliți numărul circuitelor care conțin toate nodurile grafului:

- a. exact unul    b. cel mult unul    c. nici unul    d. cel puțin unul

(Bacalaureat – Sesiunea iunie-iulie 2003)

48. Se consideră graful neorientat având nodurile notate cu 1, 2, 3, 4, 5 corespunzător liniilor matricei de adiacență alăturate. Stabiliți care dintre următoarele propoziții este adevărată.

- a. graful este conex    b. orice nouă muchie s-ar adăuga graful devine conex  
c. graful este aciclic    d. orice muchie s-ar elimina graful devine aciclic

(Bacalaureat – Simulare 2003)

49. Gradul maxim al unui nod ce se poate obține într-un graf neorientat conex cu  $n$  noduri și  $n-1$  muchii este (s-a notat cu  $[n/2]$  câtul împărțirii întregi a lui  $n$  la 2):

- a.  $n-1$     b.  $[n/2]$     c.  $n$     d. 2

(Bacalaureat – Sesiunea iunie-iulie 2003)

50. Determinați numărul total de grafuri neorientate conexes cu 3 noduri, două grafuri fiind considerate distincte dacă și numai dacă matricele lor de adiacență diferă prin cel puțin un element.

- a. 7    b. 8    c. 4    d. 2

(Bacalaureat – Sesiunea august 2003)

51. În graful neorientat dat prin matricea de adiacență alăturată numărul de componente conexe este:

- a. 0    b. 2    c. 3    d. 1

(Bacalaureat – Sesiunea iunie-iulie 2003)

52. Stabiliți care dintre următoarele matrice de adiacență corespunde unui graf conex aciclic:

- a) 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |

    b) 

|   |   |   |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 0 |

    c) 

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |

    d) 

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |

(Bacalaureat – Sesiunea august 2003)

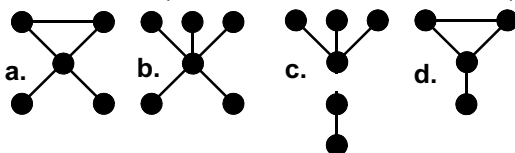
53. Orice graf neorientat cu  $n$  noduri și  $n-1$  muchii

- a. este aciclic și neconex    b. este conex dacă și numai dacă este aciclic  
c. este un arbore    d. este conex și conține un ciclu

(Bacalaureat – Simulare 2003)

54. Stabiliți care dintre grafurile alăturate este arbore:

(Bacalaureat – Simulare 2004)



## Miniproiecte:

Pentru realizarea miniproiectelor se va lucra în echipă. Fiecare miniproiect va conține:

- descompunerea problemei în subprobleme și rezolvarea fiecărei subprobleme cu ajutorul unui subprogram;
- specificățiile fiecărui modul (subprogram);
- prezentarea noțiunilor teoretice folosite pentru realizarea subprogramelor.

1. Într-un grup de  $n$  persoane s-au stabilit două tipuri de relații: de prietenie și de vecinătate. Scrieți un program care să citească matricele de adiacență ale celor două grafuri



dintr-un fișier (pe primul rând, numărul de persoane, și pe următoarele rânduri, în ordine, liniile fiecărei matrice de adiacență) și care să afișeze:

- a. persoanele care sunt și prietene și vecini (**Indicație.** Se determină muchiile din graful intersecție a celor două grafuri neorientate);
- b. cel mai mare număr de persoane care se găsesc într-un grup de vecini prieteni (**Indicație.** Se determină gradul maxim în graful intersecție a celor două grafuri.)
- c. pentru fiecare persoană, persoanele cu care nu este în relație de prietenie;
- d. cel mai mare grup de prieteni. (**Indicație.** Se determină componenta conexă care conține cele mai multe noduri).
- e. grupurile de prieteni, grupurile de vecini și grupurile de vecini prieteni. (**Indicație.** Se determină componentele conexe pentru fiecare graf.)

Pentru testarea programului formați un grup cu 10 dintre colegii voștri și descrieți, cu ajutorul a două grafuri, cele două tipuri de relații.

2. Într-un grup de  $n$  persoane s-a stabilit o relație de cunoștință: persoana  $x$  este în relație cu persoana  $y$  dacă o cunoaște pe aceasta. Relația de cunoștință nu este reciprocă. O *celebritate* este o persoană care este cunoscută de toate persoanele din grup, dar care nu cunoaște nici o persoană (este un nod destinație al grafului). Un *necunoscut* este o persoană care cunoaște toate persoanele din grup dar nu este cunoscută de nici o persoană din grup (este un nod sursă al grafului). Un *singuratic* este o persoană care cunoaște o singură persoană din grup sau este cunoscută de o singură persoană din grup (este un nod terminal al grafului). Un *străin de grup* este o persoană care nu cunoaște nici o persoană din grup și nu este cunoscută de nici o persoană din grup (este un nod izolat al grafului). Demonstrați că în grup nu poate exista decât o singură *celebritate* și un singur *necunoscut*. Scrieți un program care să citească matricea de adiacență a grafului, dintr-un fișier, și care să afișeze:

- a. dacă există o *celebritate*, să se precizeze persoana, iar dacă nu există, să se precizeze: persoana care cunoaște cele mai puține persoane și persoana care este cunoscută de cele mai multe persoane;
- b. dacă există un *necunoscut*, să se precizeze persoana, iar dacă nu există, să se precizeze: persoana care este cunoscută de cele mai puține persoane și persoana care cunoaște cele mai multe persoane;
- c. dacă există *singuratici*, să se precizeze persoanele;
- d. dacă există *străini de grup*, să se precizeze persoanele;
- e. câte persoane cunosc doar două persoane din grup și sunt cunoscute la rândul lor de trei persoane din grup (nodurile care au gradul intern egal cu 3, iar gradul extern egal cu 2);
- f. câte persoane sunt cunoscute de un număr de membri egal cu numărul de membri pe care îi cunosc (nodurile care au gradul intern egal cu gradul extern);
- g. care sunt persoanele care se cunosc reciproc (perechile de noduri între care există arce duble);
- h. relațiile în cadrul grupului după ce au fost eliminați străinii de grup și singuraticii; (**Indicație.** Se generează un subgraf al grafului inițial, prin eliminarea tuturor nodurilor izolate sau terminale.)
- i. grupul format din cele mai multe persoane care se cunosc reciproc. (**Indicație.** Se determină componenta tare conexă care conține cele mai multe noduri).

3. La graful județelor, adăugați un nod cu eticheta 0, care reprezintă exteriorul țării, și muchiile corespunzătoare, care evidențiază județele limitrofe. Creați cu un editor de texte fișierul

*judete.txt* care să conțină următoarele informații: pe primul rând, numărul de județe, pe următoarele rânduri matricea de adiacență, pe următorul rând, în ordinea etichetelor nodurilor, denumirile județelor separate prin spațiu (pentru județele care au un nume format din mai multe cuvinte, folosiți ca separator linia de subliniere), pe următorul rând numele vecinilor României, separați prin spațiu, și apoi, câte un rând, pentru fiecare județ limitrof, în care scrieți, separate prin spații, următoarele informații: eticheta nodului și vecinii, precizați prin numărul de ordine, în lista numelor. Construiți un vector în care memorați, pentru fiecare județ, un indice pentru provincia istorică din care face parte (1 – Muntenia, 2 – Moldova etc.). Scrieți un program care să citească aceste informații din fișier și să le transpună într-o implementare a grafului, adecvată problemei, și care să afișeze:

- a. județele care au cele mai multe județe vecine – se va preciza numărul maxim de vecini și numele județelor care au această proprietate;
  - b. județele care au cele mai puține județe vecine – se va preciza numărul minim de vecini și numele județelor care au această proprietate;
  - c. județele de la graniță – pentru fiecare județ se va preciza numele său și numele țărilor vecine.
  - d. subgrafurile provinciilor istorice; precizați provincia istorică ce conține cele mai multe județe, și provincia istorică ce conține cele mai puține județe.
  - e. numărul minim, respectiv numărul maxim, de județe prin care trebuie să treacă un turist care tranzitează prin România, știind că vine din Ungaria și trebuie să ajungă în Bulgaria. (**Indicație.** Se determină un lanț de lungime minimă, respectiv maximă, între două județe limitrofe, care sunt la granița cu cele două țări.)
4. Într-o zonă turistică există  $n$  localități. Între unele localități există legături directe, fie prin șosele naționale, fie prin șosele județene. Legăturile directe sunt caracterizate de lungimea drumului, măsurată în kilometri. Se vor folosi două grafuri:  $G_n=(X,U_n)$ , pentru legăturile prin șosele naționale, și  $G_j=(X,U_j)$ , pentru legăturile prin șosele județene. Cele două grafuri se vor citi din două fișiere text, care conțin, pentru fiecare legătură directă, pe câte un rând, separate prin spații, cele două etichete ale nodurilor asociate localităților și distanța dintre localități. Scrieți un program care să citească aceste informații din fișier și să le transpună într-o implementare a grafului, adecvată problemei, și care să afișeze:
- a. localitățile la care nu ajung drumuri naționale (nodurile izolate, în primul graf);
  - b. cea mai scurtă legătură directă dintre două localități – se va preciza eticheta nodurilor asociate localităților, distanța dintre localități și tipul șoselei prin care se asigură legătura;
  - c. pentru două localități precizate,  $a$  și  $b$  (etichetele nodurilor  $a$  și  $b$  se citesc de la tastatură), să se precizeze dacă există legătură directă; dacă există, să se mai precizeze tipul șoselei și distanța dintre localități;
  - d. pentru o localitate  $p$  (eticheta  $p$  a nodului se citește de la tastatură), să se afișeze toate localitățile cu care are legături directe și distanța până la aceste localități (**Indicație.** Se determină graful reuniune a celor două grafuri.);
  - e. localitatea care are cele mai multe legături directe cu alte localități – și să se afișeze localitățile cu care are legături (nodul cu gradul maxim în graful reuniune).
  - f. numărul de localități care rămân izolate dacă se distruge un pod pe care trece șoseaua care leagă două localități ale căror etichete se citesc de la tastatură.
5. Într-un munte există  $n$  grote. Fiecare grotă  $i$  se găsește la înălțimea  $h_i$  față de baza muntelui. Între unele grote există comunicare directă, prin intermediul unor tuneluri.

Există și grotte care comunică cu exteriorul. Desenați o rețea ipotetică de grotte și construiți matricea de adiacență a grafului neorientat asociat (grottele sunt nodurile, și tunelurile sunt muchiile). Înălțimile grotelor se vor memora într-un vector. Găsiți o modalitate de a evidenția grottele care comunică cu exteriorul. (**Indicație.** Adăugați la graf nodul 0, care reprezintă exteriorul muntelui.) Scrieți matricea de adiacență și vectorul cu înălțimile grotelor, în fișierul text *grote.txt*. Scrieți un program care să citească matricea de adiacență și vectorul din fișier – și care să afișeze:

- a. numărul de tuneluri de comunicare (numărul de muchii ale grafului);
- b. grottele care au legătură cu exteriorul (nodurile  $i$  pentru care există muchie cu nodul 0);
- c. grottele care comunică direct cu cele mai multe grotte (nodurile cu cel mai mare grad);
- d. grottele care nu comunică prin tuneluri cu alte grotte (nodurile izolate);
- e. cea mai înaltă grotă și cea mai joasă grotă care comunică cu exteriorul;
- f. pentru o grotă a cărei etichetă se citește de la tastatură, să se afișeze grottele cu care comunică direct, precizând pentru fiecare tunel dacă suie, coboară sau este la același nivel cu grotă.
- g. subgraful grotelor care se găsesc la o înălțime  $h$ , cu  $h_1 \leq h \leq h_2$  și câte dintre aceste grotte comunică direct cu exteriorul muntelui (valorile pentru înălțimile  $h_1$  și  $h_2$  se citește de la tastatură);
- h. grottele care vor fi inundate de un pârau subteran care izvorăște dintr-o grotă a cărei etichetă se citește de la tastatură și tunelul (sau tunelurile) prin care pâraul va ajunge în exteriorul muntelui. (**Indicație.** O grotă este inundată dacă se găsește la același nivel sau la un nivel mai jos decât o grotă în care există apă. Se transformă graful neorientat într-un graf orientat, în care arcele vor pune în evidență relația: grotă  $x$  este în relație cu grotă  $y$  dacă apa curge de la grotă  $x$  la grotă  $y$ . În acest graf, determinați toate drumurile care pornesc din grotă din care izvorăște pâraul).

6. Rețeaua de străzi dintr-un oraș este formată din străzi cu două sensuri și străzi cu sens unic de circulație. Ea poate fi reprezentată printr-un graf orientat, în care intersecțiile sunt nodurile, iar arcele – sensul de circulație pe străzile care leagă două intersecții (traficul auto). Nodurile sunt intersecții de cel puțin 3 străzi. Urmează să se modernizeze intersecțiile – pentru a stabili prioritatea în intersecții și pentru a fluidiza traficul. Pentru modernizare, se vor folosi panouri cu semne de circulație, semafoare – și se vor amenaja sensuri giratorii. Pentru fiecare stradă din care se poate intra în intersecție, se montează în intersecție un panou cu semn pentru prioritate. Pentru fiecare stradă pe care nu se poate intra din intersecție se montează în intersecție un panou cu semnul de interdicere a circulației. În toate intersecțiile vor fi montate panouri cu semne de circulație, corespunzător străzilor incidente cu intersecția. Fiecare dintre aceste mijloace de modernizare are un cost: panoul cu semn de circulație – costul  $c_1$ , semaforul – costul  $c_2$ , și sensul giratoriu – costul  $c_3$ . Pentru a stabili modul în care este modernizată fiecare intersecție, intersecțiile au fost clasificate în intersecții mici (intersecții cu 3 străzi, în care vor fi montate numai panouri cu semne de circulație), intersecții mari (intersecții cu 4 străzi, care vor fi semaforizate) și intersecții foarte mari (intersecții cu peste 4 străzi, în care se vor amenaja sensuri giratorii). Desenați o rețea ipotetică de străzi și construiți matricea de adiacență a grafului orientat asociat. Scrieți matricea de adiacență în fișierul text *strazi.txt*. Scrieți un program care să citească matricea de adiacență din fișier și care să afișeze:

- a. intersecțiile la care nu se poate ajunge din nici o altă intersecție (nodurile care nu au predecesori);
  - b. intersecțiile de la care nu se poate ajunge la o nici o altă intersecție (nodurile care nu au succesori);
  - c. dacă există intersecții fără nici o intersecție succesor sau fără nici o intersecție predecesor, să se corecteze desenul rețelei, astfel încât să nu existe asemenea intersecții (prin adăugarea unui număr minim de arce) și să se actualizeze și fișierul *strazi.txt*.
  - d. intersecțiile la care se poate ajunge direct din cele mai multe intersecții (nodurile care au cel mai mare grad intern);
  - e. intersecțiile de la care se poate ajunge direct la cele mai multe intersecții (nodurile care au cel mai mare grad extern);
  - f. numărul de străzi pe care se circulă în ambele sensuri (numărul de perechi de noduri  $i$  și  $j$  pentru care, în matricea de adiacență, elementele  $a[i][j]$  și  $a[j][i]$  sunt egale cu 1);
  - g. numărul de intersecții mici, mari și foarte mari (clasificarea nodurilor în funcție de numărul de noduri adiacente: noduri cu 3 noduri adiacente, cu 4 noduri adiacente și cu cel puțin 5 noduri adiacente);
  - h. numărul de panouri cu semne pentru prioritate care se vor folosi (suma gradelor interne ale nodurilor);
  - i. numărul de panouri cu semne pentru interzicerea sensului care se vor folosi (diferența dintre suma gradelor externe ale nodurilor și numărul de străzi cu sens dublu de circulație);
  - j. numărul de semafoare care se vor monta (suma gradelor interne ale nodurilor care au 4 noduri adiacente);
  - k. costul de modernizare necesar pentru fiecare intersecție și costul total al modernizării;
  - l. să se precizeze dacă, prin închiderea unor străzi care urmează să fie reparate, traficul auto nu este perturbat, în sensul că vor exista intersecții la care nu se mai poate ajunge – etichetele intersecțiilor care sunt legate de aceste străzi se citesc de la tastatură; (**Indicație.** Se generează graful parțial al traficului, prin eliminarea arcelor dintre nodurile precizate, și se verifică dacă există noduri care au gradul intern egal cu 0.)
  - m. să se precizeze dacă, prin închiderea acestor străzi, nu se izolează unele zone ale orașului de alte zone. (**Indicație.** Se generează graful parțial al traficului, prin eliminarea arcelor dintre nodurile precizate și se verifică dacă graful obținut este tare conex.)
7. Pe un munte există mai multe parcele cu fânețe ale sătenilor. Unele fânețe au acces direct la drumul sătesc, altele nu. Între proprietarii parcelor vecine există relație de prietenie – sau nu. Dacă doi proprietari vecini sunt prieteni, își permit unul altuia accesul pe propria parcelă. Pentru a transporta fânul, sătenii care nu au acces la drumul sătesc, trebuie să treacă peste parcelele altor săteni, ca să ajungă la el. Un proprietar este considerat izolat dacă nu are acces direct la drumul sătesc și nici nu este în relație de prietenie cu vreunul dintre vecinii lui. Se vor folosi două grafuri: unul pentru a reprezenta relația de vecinătate a fânețelor, iar altul pentru a reprezenta relația de prietenie dintre proprietari. Desenați o hartă ipotetică a fânețelor și stabiliți relații ipotetice de prietenie între vecini. Construiți matricele de adiacență ale celor două grafuri asociate. Scrieți matricele de adiacență în fișierele text *fanete.txt* și *prietenii.txt*. Scrieți un program care să citească matricele de adiacență din fișiere și care să furnizeze următoarele informații:

- a. dacă există proprietari izolați, să se afișeze lista proprietarilor vecini cu care trebuie să stabilească relații de prietenie, pentru a ajunge la drumul sătesc, și să se identifice vecinii care au acces direct la drumul sătesc;
- b. care este proprietarul cel mai neprietenos (care are cel mai mare procent de vecini cu care nu este prieten).
- c. care sunt sătenii cei mai îndepărtați de drumul sătesc (**Indicație**. Din lanțurile de lungime minimă din graful fânețelor găsiți cel mai lung lanț.)
- d. care sunt sătenii izolați. (**Indicație**. Sătenii care nu se găsesc în graful de vecinătate într-o componentă conexă în care se află și un sătean cu ieșire la drumul sătesc.)

Se citesc, din două fișiere text două matrice de adiacență a două grafuri care au același număr de noduri. Să se verifice dacă aceste matrice pot reprezenta matricele de adiacență a relației de vecinătate, respectiv a relației de prietenie pentru sătenii care au fânețe. (**Indicație**. Se verifică dacă graful cu mai puține muchii este graf parțial al celui alt graf).

8. Se analizează migrația unei specii de păsări călătoare pe perioada unui an. Migrația are două etape: migrația de toamnă, când păsările migrează din zonele reci către zonele calde (migrația de la rece la cald), și migrația de primăvară, când păsările migrează din zonele calde către zonele reci (migrația de la cald la rece). Fiecare etapă a migrației va fi reprezentată printr-un graf orientat. În graful migrației de la rece la cald, nodurile care aparțin zonei reci au numai succesori, iar nodurile care aparțin zonei calde au numai predecesori. În graful migrației de la cald la rece nodurile care aparțin zonei calde au numai succesori, iar nodurile care aparțin zonei reci au numai predecesori. Fiecărui nod  $i$  se asociază coordonatele geografice. Desenați câte o hartă ipotetică pentru fiecare etapă de migrație. Construiți, cu un editor de texte, fișierul *migratie1.txt*, care va conține pe prima linie  $n1$ , numărul de noduri ale primului graf, pe următoarele  $n1$  linii, matricea de adiacență a grafului, și apoi, pe următoarele  $n1$  linii, coordonatele geografice ale fiecărui nod de pe hartă (sunt 6 entități de informație, separate prin spațiu: două valori numerice întregi pentru latitudine, în grade și minute, și un caracter pentru emisferă, două valori numerice întregi pentru longitudine, în grade și minute, și un caracter pentru meridian). Construiți cu un editor de texte fișierul *migratie2.txt*, care va conține același tip de informații, pentru cel de al doilea graf al migrației. Scrieți un program care să citească aceste informații din fișier, să le transpună într-o implementare a grafului adecvată problemei (**recomandare** – implementare prin matrice de adiacență și vector de structuri pentru coordonatele geografice) și care să afișeze următoarele informații:

- a. dacă cele două grafuri sunt corecte (dacă  $a[i][j]=1$ , atunci  $a[j][i]=0$  – și produsul dintre gradul intern și gradul extern ale fiecărui nod trebuie să fie 0; în plus, trebuie ca nodurile din zona caldă să coincidă – în ambele grafuri);
- b. dacă în graful migrației de la rece la cald există un nod destinație al grafului – și ce semnificație are existența lui pentru migrație;
- c. dacă păsările s-au reîntors în aceleași zone, primăvara (nodurile din zona rece coincid în cele două grafuri); în caz contrar, să se specifice noile locații apărute în zona rece în care a ajuns specia respectivă;
- d. să se obțină matricea de adiacență a hărții migrației, pe întreaga perioadă a anului, prin reuniunea celor două grafuri (dacă după migrația de primăvară au apărut noi locații pe hartă în zona rece, se vor adăuga la primul graf ca noduri izolate).

# Anexă

## Tipuri de date specifice pentru adresarea memoriei

Pentru a putea manipula o dată, programul trebuie să identifice zona de memorie în care este stocată. Identificarea acestei zone se poate face atât prin numele datei (**a**), cât și prin adresa la care este memorată (**adr**). Pentru a putea manipula datele cu ajutorul adreselor de memorie, în limbajul C++ sunt implementate următoarele tipuri de date:

- tipul **pointer** sau **adresă**;
- tipul **referință**.

**Pointerul** este o **variabilă de memorie în care se memorează o adresă de memorie**.

**Observație.** Un pointer **se asociază întotdeauna unui tip de dată** (de exemplu: **int**, **char**, **float** etc.) numit **tip de bază**. Se spune că un pointer **indică** o zonă de memorie care conține o dată care are tipul de bază. Este necesar să se asocieze pointerului un tip de dată deoarece el memorează numai o adresă și, pentru a manipula conținutul unei zone de memorie, compilatorul trebuie să cunoască dimensiunea zonei de memorie care începe de la acea adresă și semnificația conținutului, adică ce algoritm de decodificare trebuie să folosească pentru a transforma secvența de biți (din acea zonă de memorie) într-o dată.

Declararea unei variabile de tip pointer se poate face prin instrucțiunea declarativă:

`tip_dată *nume_variabilă;`

tipul datei stocate la adresa memorată în pointer (**tipul de bază**)

numele variabilei de tip pointer în care se memorează adresa unei variabile de memorie care are tipul de bază

operatorul de **referință**

**Exemplu:**

```
char *p;
int *q;
float *r;
```

S-au definit trei variabile de tip adresă (pointeri): **p** către tipul **char**, **q** către tipul **int** și **r** către tipul **float**. Numele acestor variabile de memorie sunt **p**, **q** și **r**, iar tipul de bază al fiecărei variabile de tip pointer este **char**, **int** respectiv **float**. Așadar, **p** este o variabilă de tip adresă a unei locații ce conține un caracter care ocupă 1 octet, **q** este o variabilă de tip adresă a unei locații ce conține o dată numerică întreagă care ocupă 2 octeți, iar **r** este o variabilă de tip adresă a unei locații ce conține o dată numerică reală care ocupă 4 octeți.

O **constantă simbolică de tip adresă** este un **pointer al cărui conținut nu poate fi modificat**.

**Exemplu.** **Constanta 0** care semnifică „adresă nulă” (adresă inexistentă sau adresă nealocată) este utilă în unele cazuri pentru inițializarea valorii unui pointer. În locul constantei numerice literale 0, se poate folosi **constantă simbolică predefinită NULL**.

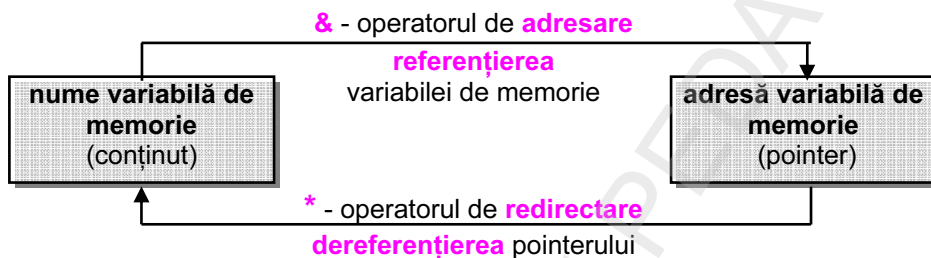
Pe variabilele de tip pointeri se pot aplica următoarele tipuri de operatori:

- **operatori specifici**,
- **operatorul de atribuire**,
- **operatori aritmetici**,
- **operatori relaționali**.



**Operatorii specifici** sunt:

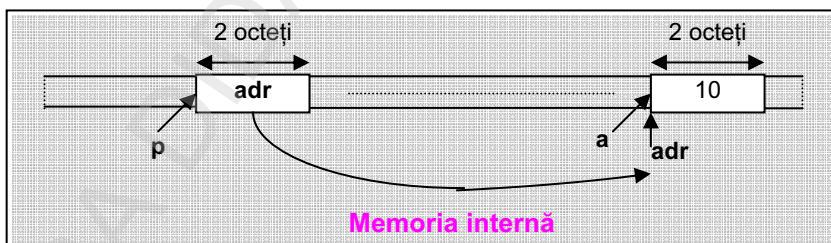
- **Operatorul &** – operatorul de **adresare**. Se aplică pe o variabilă de memorie sau un element de tablou de memorie și furnizează adresa variabilei de memorie, respectiv a elementului de tablou. Rezultatul obținut prin aplicarea acestui operator este de tip pointer. Operația se numește **referențierea** unei variabile de memorie.
- **Operatorul \*** – operatorul de **redirectare**. Se aplică pe o variabilă de tip pointer și furnizează valoarea variabilei de memorie care se găsește la adresa memorată în pointer. Rezultatul obținut prin aplicarea acestui operator este de tipul datei asociate pointerului. Operația se numește **dereferențierea** unui pointer.



**Exemplu:**

```
int a=10, *p=&a;
cout<<*p<<endl<<a<<endl;
cout<<p<<endl<<&a<<endl;
```

S-a definit o variabilă de memorie de tip **int** (**a**) și o variabilă de tip pointer către tipul **int** (**\*p**) căreia i s-a atribuit ca valoare adresa variabilei **a**. În memoria internă se vor alocă două zone de memorie: una de 2 octeți pentru variabila de memorie identificată prin numele **a**, în care se păstrează o valoare numerică întreagă, și una de 2 octeți pentru variabila pointer identificată prin numele **p**, în care se păstrează o adresă de memorie (în exemplu, adresa variabilei de memorie **a**). Prin instrucțiunea **cout<<\*p**; se afișează conținutul variabilei de memorie **a**. Referirea la această variabilă de memorie se face nu prin numele variabilei, ci prin operatorul de redirectare **\*** aplicat pe variabila pointer **p** în care este memorată adresa variabilei **a**. Această instrucțiune are același efect ca și instrucțiunea **cout<<a**; . Prin instrucțiunea **cout<<p**; se afișează o constantă hexazecimală care reprezintă o adresă de memorie (adresa **adr** la care este memorată variabila **a**). Această instrucțiune are același efect ca și instrucțiunea **cout<<&a**;



## Operatorul de atribuire

Unei variabile de tip pointer i se poate atribui numai o valoare care reprezintă o adresă de memorie. Ea poate fi exprimată prin:

- O **adresă de memorie** obținută prin **aplicarea operatorului de adresare** pe o variabilă de memorie definită anterior, de același tip cu tipul de bază al pointerului. După operația de atribuire, variabila de tip pointer va adresa zona de memorie în care este stocată variabila de memorie.



- O altă **variabilă de tip pointer** care se referă la același tip de bază. După operația de atribuire, cele două variabile de tip pointer vor adresa aceeași zonă de memorie.
- O **constantă de tip adresă** de același tip cu tipul de bază al pointerului:
  - **Numele unui tablou de memorie** este o constantă de tip adresă. După operația de atribuire, variabila de tip pointer va adresa zona de memorie în care este stocat primul element al tabloului – elementul cu indicele 0.
  - **Constanta 0** sau constanta simbolică predefinită **NULL** care corespunde **pointerului zero** poate fi atribuită oricărui tip de pointer.
- **Rezultatul unei expresii construite cu operanzi de tip adresă** a unor date definite anterior și care aparțin tipului de bază al pointerului.

### Operatorii aritmetici

Operațiile aritmetice permise asupra pointerilor sunt:

- **Adunarea și scăderea unei constante** – operatorii **+** și **-**.
- **Incrementarea și decrementarea unui pointer** – operatorii **++** și **--**.
- **Scăderea a doi pointeri de același tip** – operatorul **-**.

#### Atenție

Toate operațiile aritmetice cu pointeri către un tip de bază se execută considerând **unitatea egală** cu **sizeof(tip\_bază)**, adică egală cu numărul de octeți necesari pentru a memora o dată care are tipul **tip\_bază**. De exemplu, incrementarea unui pointer către tipul **int** va considera unitatea egală cu 2 (numărul de octeți necesari pentru reprezentarea tipului **int**) și noua adresă se va obține prin adunarea constantei 2 la adresa inițială memorată de pointer.

Operatorul pentru **adunarea sau scăderea unei constante** este:

**p + k sau p - k**

unde **p** este un pointer către tipul **tip\_baza** care memorează o adresă **adr**, iar **k** o constantă. Rezultatul este o adresă care se calculează astfel: **adr+k\*sizeof(tip\_baza)**, respectiv **adr-k\*sizeof(tip\_baza)**.

**Observație:** Operația de adunare este comutativă: **p+k = k+p**.

Operatorul pentru **incrementarea și decrementarea unui pointer** este:

**p++ sau p--**

unde **p** este un pointer către tipul **tip\_baza** care memorează o adresă **adr**. Rezultatul este o adresă care se calculează astfel: **adr+sizeof(tip\_baza)**, respectiv **adr-sizeof(tip\_baza)**.

Operatorul pentru **scăderea a doi pointeri** este:

**p - q**

unde **p** și **q** sunt doi pointeri către același tip de bază, care indică elementele aceluiasi tablou de memorie, adresa memorată de **p** fiind mai mare decât cea memorată de **q** (**p** indică un element de tablou situat în memorie după elementul indicat de pointerul **q**). Rezultatul este un număr întreg care reprezintă numărul de elemente aflate între **p** și **q**.

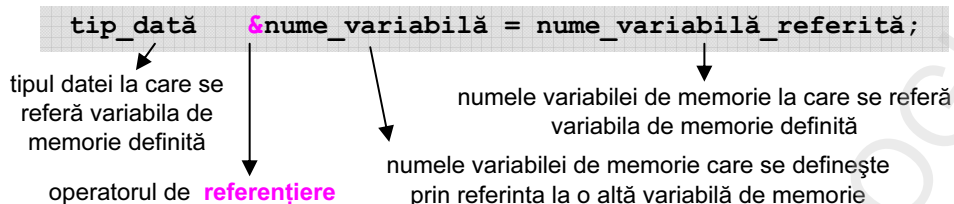
**Operatorii relaționali** sunt:

- **Operatorii de inegalitate:** **<**, **>**, **<=** și **>=**.
- **Operatorii de egalitate:** **==** și **!=**.

Evaluarea unui operator relațional aplicat pe doi pointeri **p** și **q** se face prin analiza diferenței **p-q**.

**Tipul de dată referință** permite folosirea mai multor identificatori pentru aceeași variabilă de memorie.

Declararea unei variabile de tip referință se poate face prin instrucțiunea declarativă:

**Exemplu:**

```
int a=10;
int &b=a;
sau
```

```
int a=10, &b=a;
```

Variabila de memorie **b** este o referință către variabila de memorie **a**. Aceasta înseamnă că, prin referință, variabilei de memorie (**a**) i s-a mai dat un nume (**b**). Cele două variabile de memorie (**a** și **b**) au aceleași atribute (adresă de memorie, lungime, valoare, tip etc.), cu excepția numelui. Chiar dacă în program s-au declarat două variabile de memorie (**a** și **b**), se lucrează cu o singură zonă de memorie, care poate fi identificată cu două nume diferite, iar următoarele două instrucțiuni vor afișa aceleași valori:

```
cout<<"a = "<<a<<"adresa lui a= "<<&a<<endl;
cout<<"b = "<<b<<"adresa lui b= "<<&b;
```

Orice operație de modificare a valorii variabilei referite prin numele **b**, va modifica valoarea variabilei identificate prin numele **a**, deoarece ambele nume de variabile se referă la conținutul aceleiași zone de memorie. Următoarele două instrucțiuni vor afișa aceleași valori:

```
b=15;
cout<<"a = "<<a<<"adresa lui a= "<<&a<<endl;
cout<<"b = "<<b<<"adresa lui b= "<<&b;
```

Tipul de dată referință este util în transmiterea datelor între modulul apelant și subprogram, prin intermediul parametrilor de comunicație. Folosind **transferul parametrului prin referință**, în momentul apelării subprogramului, în stivă este încărcată adresa de memorie la care se găsește valoarea parametrului. Subprogramul va lucra direct în zona de memorie în care se găsește data. Așadar, atât modulul apelant cât și subprogramul lucrează asupra aceleiași date, și orice modificare a valorii acestui parametru făcută în subprogram se va reflecta și în modulul apelant. La terminarea execuției subprogramului, este eliberată din stivă zona în care este memorată adresa parametrului.

## Cuprins

|                                                                                                  |           |
|--------------------------------------------------------------------------------------------------|-----------|
| <b>1. Tehnici de programare .....</b>                                                            | <b>3</b>  |
| <b>1.1. Subprogramele.....</b>                                                                   | <b>3</b>  |
| 1.1.1. Definiția subprogramului .....                                                            | 3         |
| 1.1.1.1. Necesitatea folosirii subprogramelelor .....                                            | 4         |
| 1.1.1.2. Terminologie folosită pentru subprograme .....                                          | 5         |
| 1.1.1.3. Avantajele folosirii subprogramelelor .....                                             | 6         |
| 1.1.2. Parametrii de comunicare .....                                                            | 6         |
| 1.1.3. Elementele subprogramului .....                                                           | 7         |
| 1.1.4. Clasificarea subprogramelelor .....                                                       | 8         |
| 1.1.4.1. Clasificarea în funcție de modalitatea de apel.....                                     | 8         |
| 1.1.4.2. Clasificarea în funcție de autor.....                                                   | 10        |
| 1.1.5. Reguli pentru construirea subprogramelelor C++ .....                                      | 11        |
| 1.1.5.1. Definiția subprogramului.....                                                           | 11        |
| 1.1.5.2. Prototipul subprogramului .....                                                         | 13        |
| 1.1.5.3. Activarea (apelul) subprogramului.....                                                  | 13        |
| 1.1.5.4. Parametrii de comunicare .....                                                          | 14        |
| 1.1.5.5. Utilizarea stivei de către subprograme .....                                            | 16        |
| 1.1.6. Transferul de parametri între subprograme .....                                           | 18        |
| 1.1.7. Clasificarea variabilelor de memorie .....                                                | 22        |
| 1.1.7.1. Durata de viață a variabilelor de memorie.....                                          | 23        |
| 1.1.7.2. Domeniul de vizibilitate al identificatorilor.....                                      | 23        |
| 1.1.8. Alegerea modului de implementare a subprogramului.....                                    | 27        |
| 1.1.9. Tablourile de memorie și subprogramele.....                                               | 31        |
| 1.1.10. Dezvoltarea programelor .....                                                            | 32        |
| 1.1.11. Subprogramele de sistem .....                                                            | 38        |
| <b>Evaluare .....</b>                                                                            | <b>39</b> |
| <b>1.2. Recursivitatea .....</b>                                                                 | <b>46</b> |
| 1.2.1. Definiția procesului recursiv .....                                                       | 46        |
| 1.2.2. Reguli pentru construirea unui subprogram recursiv .....                                  | 50        |
| 1.2.3. Variabilele locale și subprogramele recursive.....                                        | 51        |
| 1.2.4. Implementarea recursivă a algoritmilor elementari .....                                   | 53        |
| 1.2.4.1. Algoritmul pentru determinarea valorii minime (maxime) .....                            | 53        |
| 1.2.4.2. Algoritmul pentru calculul c.m.m.d.c. a două numere.....                                | 53        |
| 1.2.4.3. Algoritmi pentru prelucrarea cifrelor unui număr .....                                  | 54        |
| 1.2.4.4. Algoritmul pentru verificarea unui număr dacă este prim .....                           | 56        |
| 1.2.4.5. Algoritmi pentru determinarea divizorilor unui număr.....                               | 58        |
| 1.2.4.6. Algoritmi pentru conversia între baze de numerație .....                                | 59        |
| 1.2.5. Implementarea recursivă a algoritmilor pentru prelucrarea<br>tablourilor de memorie ..... | 59        |
| 1.2.6. Recursivitatea în cascadă.....                                                            | 64        |
| 1.2.7. Recursivitatea directă și indirectă .....                                                 | 66        |
| 1.2.8. Avantajele și dezavantajele recursivității .....                                          | 68        |
| <b>Evaluare .....</b>                                                                            | <b>71</b> |
| <b>1.3. Analiza algoritmilor.....</b>                                                            | <b>77</b> |
| <b>1.4. Metode de construire a algoritmilor.....</b>                                             | <b>79</b> |
| <b>1.5. Metoda Backtracking.....</b>                                                             | <b>80</b> |
| 1.5.1. Descrierea metodei Backtracking .....                                                     | 80        |
| 1.5.2. Implementarea metodei Backtracking.....                                                   | 84        |
| 1.5.3. Probleme rezolvabile prin metoda Backtracking .....                                       | 88        |

|                                                                          |            |
|--------------------------------------------------------------------------|------------|
| 1.5.3.1. Generarea permutărilor .....                                    | 88         |
| 1.5.3.2. Generarea produsului cartezian .....                            | 90         |
| 1.5.3.3. Generarea aranjamentelor .....                                  | 93         |
| 1.5.3.4. Generarea combinărilor .....                                    | 95         |
| 1.5.3.5. Generarea tuturor partițiilor unui număr natural .....          | 97         |
| 1.5.3.6. Generarea tuturor partițiilor unei mulțimi .....                | 100        |
| 1.5.3.7. Generarea tuturor funcțiilor surjective .....                   | 101        |
| 1.5.3.8. Problema celor n dame .....                                     | 103        |
| <b>1.6. Metoda „Divide et Impera“ .....</b>                              | <b>104</b> |
| 1.6.1. Descrierea metodei „Divide et Impera“ .....                       | 104        |
| 1.6.2. Implementarea metodei „Divide et Impera“ .....                    | 105        |
| 1.6.3. Căutarea binară .....                                             | 113        |
| 1.6.4. Sortarea prin interclasare (MergeSort) .....                      | 115        |
| 1.6.5. Sortarea rapidă (QuickSort) .....                                 | 116        |
| 1.6.6. Problema turnurilor din Hanoi .....                               | 119        |
| <b>Evaluare .....</b>                                                    | <b>121</b> |
| <b>2. Implementarea structurilor de date .....</b>                       | <b>125</b> |
| <b>2.1. Datele prelucrate de algoritmi .....</b>                         | <b>125</b> |
| <b>2.2. Tabloul de memorie bidimensional (matricea) .....</b>            | <b>129</b> |
| 2.2.1. Implementarea tabloului bidimensional în C++ .....                | 132        |
| 2.2.2. Algoritmi pentru prelucrarea tablourilor bidimensionale .....     | 133        |
| 2.2.2.1. Algoritmi pentru parcurgerea elementelor unei matrice .....     | 133        |
| 2.2.2.2. Algoritmi pentru prelucrarea matricelor pătrate .....           | 139        |
| <b>Evaluare .....</b>                                                    | <b>142</b> |
| <b>2.3. Șirul de caractere .....</b>                                     | <b>146</b> |
| 2.3.1. Implementarea șirului de caractere în limbajul C++ .....          | 146        |
| 2.3.2. Citirea și scrierea șirurilor de caractere .....                  | 148        |
| 2.3.3. Algoritmi pentru prelucrarea șirurilor de caractere .....         | 151        |
| 2.3.3.1. Prelucrarea a două șiruri de caractere .....                    | 154        |
| 2.3.3.2. Prelucrarea unui șir de caractere .....                         | 158        |
| 2.3.3.3. Prelucrarea subșirurilor de caractere .....                     | 163        |
| 2.3.3.4. Conversii între tipul șir de caractere și tipuri numerice ..... | 173        |
| <b>Evaluare .....</b>                                                    | <b>178</b> |
| <b>2.4. Înregistrarea .....</b>                                          | <b>181</b> |
| 2.4.1. Implementarea înregistrării în limbajul C++ .....                 | 183        |
| 2.4.1.1. Declararea variabilei de tip înregistrare .....                 | 183        |
| 2.4.1.2. Accesul la câmpurile înregistrării .....                        | 185        |
| 2.4.2. Înregistrări imbricate .....                                      | 187        |
| 2.4.3. Tablouri de înregistrări .....                                    | 192        |
| <b>Evaluare .....</b>                                                    | <b>194</b> |
| <b>2.5. Lista .....</b>                                                  | <b>198</b> |
| 2.5.1. Implementarea listelor în limbajul C++ .....                      | 200        |
| 2.5.1.1. Implementarea prin alocare secvențială .....                    | 201        |
| 2.5.1.2. Implementarea prin alocare înlănțuită .....                     | 201        |
| 2.5.2. Clasificarea listelor .....                                       | 204        |
| 2.5.3. Algoritmi pentru prelucrarea listelor generale .....              | 205        |
| 2.5.3.1. Inițializarea listei .....                                      | 206        |
| 2.5.3.2. Alocarea memoriei .....                                         | 206        |
| 2.5.3.3. Crearea listei .....                                            | 207        |
| 2.5.3.4. Adăugarea primului nod la listă .....                           | 207        |

|                                                                         |            |
|-------------------------------------------------------------------------|------------|
| 2.6.3.5. Adăugarea unui nod la listă.....                               | 207        |
| 2.5.3.6. Parcurgerea listei .....                                       | 210        |
| 2.5.3.7. Căutarea unui nod în listă .....                               | 210        |
| 2.5.3.8. Căutarea succesorului și a predecesorului unui nod.....        | 211        |
| 2.5.3.9. Eliminarea unui nod din listă .....                            | 211        |
| 2.5.3.10. Prelucrarea listelor .....                                    | 213        |
| 2.5.4. Algoritmi pentru prelucrarea stivelor .....                      | 222        |
| 2.5.4.1. Inițializarea stivei.....                                      | 224        |
| 2.5.4.2. Adăugarea unui nod la stivă.....                               | 224        |
| 2.5.4.3. Extragerea unui nod din stivă.....                             | 224        |
| 2.5.4.3. Prelucrarea stivei.....                                        | 224        |
| 2.5.5. Algoritmi pentru prelucrarea cozilor .....                       | 227        |
| 2.5.5.1. Inițializarea cozii.....                                       | 230        |
| 2.5.5.2. Adăugarea unui nod la coadă .....                              | 230        |
| 2.5.5.3. Extragerea unui nod din coadă .....                            | 230        |
| 2.5.5.4. Prelucrarea cozii.....                                         | 231        |
| <b>Evaluare .....</b>                                                   | <b>232</b> |
| <b>2.6. Graful .....</b>                                                | <b>236</b> |
| 2.6.1. Definiția matematică a grafului.....                             | 236        |
| 2.6.2. Graful neorientat .....                                          | 238        |
| 2.6.2.1. Terminologie .....                                             | 238        |
| 2.6.2.2. Gradul unui nod al grafului neorientat .....                   | 240        |
| 2.6.3. Graful orientat .....                                            | 242        |
| 2.6.3.1. Terminologie .....                                             | 242        |
| 2.6.3.2. Gradele unui nod al grafului orientat .....                    | 244        |
| 2.6.4. Reprezentarea și implementarea grafului .....                    | 246        |
| 2.6.4.1. Reprezentarea prin matricea de adiacență .....                 | 246        |
| 2.6.4.2. Reprezentarea prin lista muchiilor (arcelor) .....             | 252        |
| 2.6.4.3. Reprezentarea prin lista de adiacență (listele vecinilor)..... | 256        |
| 2.6.5. Grafuri speciale.....                                            | 261        |
| 2.6.5.1. Graful nul.....                                                | 261        |
| 2.6.5.2. Graful complet.....                                            | 261        |
| 2.6.6. Grafuri derivate dintr-un graf.....                              | 263        |
| 2.6.6.1. Graful parțial.....                                            | 264        |
| 2.6.6.2. Subgraful.....                                                 | 266        |
| 2.6.7. Conexitatea grafurilor.....                                      | 269        |
| 2.6.7.1. Lanțul .....                                                   | 269        |
| 2.6.7.2. Ciclul .....                                                   | 273        |
| 2.6.7.3. Drumul.....                                                    | 276        |
| 2.6.7.4. Circuitul .....                                                | 277        |
| 2.6.7.5. Graful conex.....                                              | 278        |
| 2.6.7.6. Graful tare conex.....                                         | 282        |
| 2.6.8. Parcurgerea unui graf .....                                      | 286        |
| 2.6.8.1. Parcurgerea în lățime – Breadth First .....                    | 286        |
| 2.6.8.2. Parcurgerea în adâncime – Depth First .....                    | 289        |
| <b>2.7. Arborele .....</b>                                              | <b>293</b> |
| 2.7.1. Definiția arborelui.....                                         | 293        |
| 2.7.2. Arborele parțial .....                                           | 295        |
| <b>Evaluare .....</b>                                                   | <b>296</b> |
| <b>Anexă .....</b>                                                      | <b>306</b> |