Metoda Backtracking

Metoda backtracking se aplică algoritmilor pentru rezolvarea următoarelor tipuri de probleme:

Fiind date n mulţimi S_1, S_2, ... S_n, fiecare având un număr nrs_i de elemente, se cere găsirea elementelor vectorului X =(x_1, x_2, ... x_n) ϵ S=S_1xS_2x…S_n, astfel încât să fie îndeplinită o anumită relaţie φ(x_1, x_2, … ,x_n) între elementele sale.

Relaţia φ(x_1, x_2, … ,x_n) se numeşte relaţie internă (condiție internă), mulţimea S=S_1xS_2x…S_n se numeşte spaţiul soluţiilor posibile, iar vectorul X se numeşte soluţia rezultat.

Metoda backtracking determină toate soluţiile rezultat ale problemei. Dintre acestea se poate alege una care îndeplineşte în plus o altă condiţie.

Această metodă se foloseşte în rezolvarea problemelor care îndeplinesc simultan următoarele condiţii:

- mulţimile S_1, S_2, ... S_n sunt mulţimi finite, iar elementele lor se consideră că se află într-o relaţie de ordine bine stabilită (de regulă sunt termenii unei progresii aritmetice);
- nu se dispune de o altă metodă de rezolvare, mai rapidă;
- x_1, x_2, …, x_n pot fi la rândul lor vectori;
- S_1, S_2, ... S_n pot fi identice.

Metoda backtracking elimină generarea tuturor celor $\prod_{i=1}^{n} \mathrm{nr}\, s_i$ posibilităţi din spaţiul soluţiilor posibile (adică a produsului cartezian al celor n mulțimi). În acest scop la generarea vectorului X, se respectă următoarele condiţii:

a) x_k primeşte valori numai dacă x_1, x_2, ... ,x_{k-1} au primit deja valori;

b) după ce se atribuie o valoare lui x_k, se verifică relaţia (condiția) numită de continuare φ`($x_1$, $x_2$, … ,$x_k$) care stabileşte situaţia în care are sens să se treacă la calculul lui $x_{k+1}$. Neîndeplinirea condiţiei φ` exprimă faptul că oricum am alege x_{k+1}, x_{k+2}, ... ,x_n nu se ajunge la soluţia rezultat.

În caz de neîndeplinire a condiţiei φ`($x_1$, $x_2$, … ,$x_k$), se alege o nouă valoare pentru $x_k$ ϵ $S_k$ dintre cele nealese, şi se reia verificarea condiţiei φ`.

Dacă mulţimea de valori x_k s-a epuizat, se revine la alegerea altei valori pentru x_{k-1} dintre cele nealese ş.a.m.d. Această micşorare a lui k dă numele metodei, ilustrând faptul că atunci când nu se poate avansa se urmăreşte înapoi secvenţa curentă din soluţia posibilă. Între condiţia internă şi cea de continuare există o strânsă legătură. Generarea soluțiilor se termină după ce au fost testate toate valorile din S_1.

Stabilirea optimă a condiţiilor de continuare reduce mult numărul de calcule.

Observaţie: metoda Backtracking are ca rezultat obţinerea tuturor soluţiilor problemei. În cazul în care se cere o sigură soluţie se poate forţa oprirea, atunci când a fost găsită.

orice soluţie se generează sub formă de vector.

Vom considera că generarea soluţiilor se face intr-o stivă. Astfel, $x_1 \in S_1$ se va găsi pe primul nivel al stivei, $x_2 \in S_2$ se va găsi pe al doilea nivel al stivei,... $x_k \in S_k$ se va găsi pe nivelul k al stivei:

Stiva ... $X_k \in S_k$ … … $X_2 \in S_2$ $X_1 \in S_1$	1. pentru generarea permutărilor mulţimii {1,2.....n}, orice nivel al stivei va lua valori de la 1 la n. 2. odată ales un element, se verifică condițiile de continuare (altfel spus, se verifică dacă elementul este valid). 3. dacă k=n+1, atunci s-a obținut o soluție posibilă care poate fi si soluție rezultat în funcție de condițiile problemei. 4. ***Observaţie:*** Problemele rezolvate cu această metodă necesită un timp îndelungat. Din acest motiv, este bine să utilizăm metoda numai atunci când nu avem la dispoziţie un alt algoritm mai eficient.

Sub formă recursivă, algoritmul backtracking poate fi redat astfel:

```
#define nmax ...//numărul maxim de mulțimi
..../* se consideră declarate global vectorii care mulţimile S_i şi numărul lor de elemente nrs_i */
int x[nmax],n,k,nrs[nmax];

void citire(){............}
void afisare(){............}
int valid(int k){ return (φ(x[1], x[2], ..., x[k])==1);}
int soluție(int k){....}//de exemplu {return k==n+1}
void backtracking_recursiv(int k)
{if(soluție(k)) afisare(x,n); /* afişarea sau eventual prelucrarea soluţiei rezultat */
  else { int i;
         for (i=1;i<=nrs[k];i++)
          {x[k]=S_k[i];       /* al i-lea element din mullţimea S_k */
           if (valid(k))backtracking_recursiv(k+1);
          }
        }
}
int main(){ citire(); backtracking_recursiv(1); return 0;}
```

Problemele care se rezolvă prin metoda backtracking pot fi împărţite în mai multe grupuri de probleme cu rezolvări asemănătoare, in funcţie de modificările pe care le vom face în algoritm.

Principalele grupuri de probleme sunt:

G1) probleme în care vectorul soluţie are lungime fixă şi fiecare element apare o singură dată în soluţie;

G2) probleme în care vectorul soluţie are lungime variabilă şi fiecare element poate să apară de mai multe ori în soluţie;

G3) probleme în plan, atunci când spaţiul în care ne deplasăm este un tablou bidimensional (backtracking generalizat).

Cele mai cunoscute probleme din G1 sunt:

1. ***Generarea produsului cartezian a n mulțimi***

Se consideră n mulţimi finite S_1, S_2, ... S_n, de forma {1,2..,s_n}. Să se genereze produsul cartezian al acestor mulţimi.

Indicații de rezolvare:

Am considerat mulţimile de forma {1,2.., s_n} pentru a simplifica problema, în special la partea de citire si afişare, algoritmul de generare rămânând nemodificat.

Identificăm următoarele particularităţi şi condiţii:

- vectorul soluție: $X=(x_1, x_2, ... x_n) \in S_1xS_2x…S_n$,
- Fiecare element $X_k \in S_k$
- Nu există condiţii interne pentru vectorul soluţie. Nu are funcție de validare.
- Obţinem soluţia când s-au generat n valori.
- Avem soluție dacă k=n+1

```
//generare prod cartezian
#include <iostream>
#include <fstream>
using namespace std;
int x[50], n,k,s[50][100], card[50];
ifstream f("fis.in");
void citeste()
{   int i,j; f>>n;
    for(i=1;i<=n;i++)
    {   f>>card[i];
        for(j=1;j<=card[i];j++) f>>s[i][j];}}
void scrie()
{    int i;cout<<endl;
     for(i=1;i<=n;i++) cout<<s[i][x[i]]<<" ";}
int solutie(int k) { return(k==n+1);}

void back(int k)
{    if(solutie(k))scrie();
     else
      for(int i=1;i<=card[k];i++)
        {x[k]=i;
          // if(valid(k)) nu este cazul
        back(k+1);}
}
int main()
{citeste(); back(1); return 0;}
```

2. *Generarea submulţimilor unei mulţimi*

Generarea submulţimilor unei mulţimi A cu n elemente se poate face cu ajutorul algoritmului de generare a combinărilor, apelându-l repetat cu valorile 1, 2, ..., n pentru a genera submulţimile cu un element, apoi cele cu două elemente, apoi cu 3 elemente etc.
Această modalitate de rezolvare este şi mai complicată şi mai puţin eficientă decât următoarea, care se bazează pe generarea produsului cartezian {0,1}x{0,1}x…{0,1} de n ori. Această a doua metodă este eficientă deoarece generează 2^n soluţii (=nr. de submulţimi ale unei mulţimi cu n elemente). Fiecare element al produsului cartezian reprezintă câte un vector caracteristic al unei submulțimi din A.
Aşadar, generăm toți vectorii caracteristici x cu n elemente, cu valorile 0 şi 1. Pentru fiecare vector soluţie parcurgem soluţia şi afişăm elementele din mulţimea A cărora le corespund valorile 1 în x.
Astfel, pentru combinaţia 001011 vom afişa elementele de pe poziţiile 3, 5 şi 6 din mulţimea iniţială.

3. *Generarea permutărilor unei mulţimi*

Se dă o mulţime cu n elemente A={a1,a2,…,an}. Se cere să se genereze si să se afişeze toate permutările ei. Altfel spus, se cere să se afişeze toate modurile în care se pot așeza elementele mulţimii A.

Folosim pentru generare mulţimea S={1,2,…,n}. Spațiul soluțiilor posibile este S^n. Un vector X϶S^n este o solutie rezultat dacă x[i]≠x[j] si x[i]϶{1,2,...,n} (condiíile interne). La pasul k:

- x[k] ϵ {1,2,…,n};
- valid (k) : x[k]≠x[1],x[2],...,x[k-1]
- solutie(k) : k=n+1.

Se pot identifica mai multe modalităţi de a verifica dacă elementul x[k] a fost plasat deja în vectorul soluţie. Cele mai importante două sunt:

- parcurgerea elementelor deja generate pentru a verifica daca x[k] apare sau nu între ele;
- folosirea unui vector cu n elemente în care vom avea valori 0 sau 1 corespunzătoare elementelor mulţimii iniţiale. Valoarea 1 va preciza faptul că elementul de pe poziţia corespunzătoare a fost plasat anterior în vectorul soluţie, iar valoarea 0 că nu.

Corespunzător acestor două moduri de a verifica dacă un element a mai fost sau nu plasat în vectorul soluţie, avem 2 moduri de generare a permutărilor.

4. *Generarea aranjamentelor*

Generăm aranjamentele unei mulţimi atunci când ni se cer toate modurile de a alege m elemente distincte dintre cele n ale mulţimii (m<n).

Această problemă se rezolvă foarte uşor folosind metodele de generarea permutărilor. Singurele modificări presupun citirea numărului m, modificarea condiţiei de soluţie, care va fi k=m în loc de k=n şi a numărului de elemente afişate.

Folosim pentru generare mulţimea S={1,2,…,n}. Spațiul soluțiilor posibile este S^m. Un vector X϶S^m este o solutie rezultat dacă x[i]≠x[j] si x[i]϶{1,2,...,n} (condiíile interne). La pasul k:

- x[k] ϵ {1,2,…,n};
- valid (k) : x[k]≠x[1],x[2],...,x[k-1]
- soluție(k) : k=m+1.

5. Generarea submulţimilor cu m elemente ale unei mulţimi (combinări)

Generăm combinărilor unei mulţimi presupune o condiţie suplimentară faţă de permutări sau aranjamente. Acest lucru se datorează faptului că generarea combinărilor presupune alegerea în ordine strict crescătoare a elementelor care compun vectorul soluţie. Astfel, condiţia de continuare, sau de validare a unui element este aceea că el trebuie să fie strict mai mare decât cel plasat anterior. În acest mod asigurăm faptul că elementele nu se vor repeta şi că vor fi generate în ordine strict crescătoare. Trebuie, însă, să avem grijă să nu punem această condiţie si asupra primului element din vectorul soluţie, deoarece el nu are cu cine să fie comparat sau să inițializăm X[0]=0.

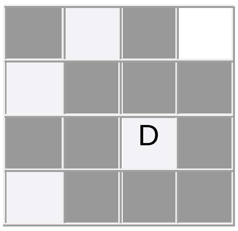
O optimizare a algoritmului de generare a combinărilor se poate obţine pornind instrucţiunea for pentru plasarea unui element de la valoare următoare valorii generate anterior.Astfel nu mai trebuie să verificăm dacă elementul Xk este mai mare ca Xk-1.

Folosim pentru generare mulţimea S={1,2,…,n}. Spațiul soluțiilor posibile este S^m. Un vector X϶S^m este o solutie rezultat dacă x[i]≠x[j] si x[i]϶{1,2,...,n} (condițiile interne). La pasul k:

- x[0]=0
- x[k] ϵ { x[k-1]+1,.….,n }; //optim x[k] ϵ { x[k-1]+1,.….,n-m+k}
- valid (k) : fiecare valoare aleasă x[k] este validă; nu necesita validare
- soluție(k) : k=m+1.
-

6. Aranjarea a n regine pe o tablă de şah de dimensiune nxn fără ca ele să se atace.

Dându-se o tablă de şah de dimensiune nxn (n>3) să se aranjeze pe ea n regine fără ca ele să se atace. Reamintim că o regină atacă linia, coloana şi cele 2 diagonale pe care se află. În figura de mai jos celulele colorare mai închis sunt atacate de regina poziţionată unde indică litera “D”.



Se plaseaza câte o regină pe fiecare linie.
Condiţia de a putea plasa o regină pe poziţia k presupune verificarea ca să nu se atace cu nici una dintre celelalte k-1 regine deja plasate pe tabla. Dacă pe poziţia k din vectorul X punem o valoare ea va reprezenta coloana pe care se plasează pe tablă regina k.

- x[k] ϵ { 1,2,.….,n };
- Validare(k): x[i]≠x[k] şi |k-i|≠|x[k]-x[i]| cu i=2,...,k-1.
- Soluție: k=n+1

7. *Generarea tuturor secvenţelor de n (par) paranteze care se închid corect.*

Să se genereze toate șirurile de n parateze rotunde închise corect.

Exemplu:

- Pt n=4: (()) ; ()()
- Pt n=6: ((())) ; ()(()) ; ()()() ; (()()) ; (())()

Notam cu 1 paranteza stânga si cu 2 paranteza dreaptă. Un vectorul soluție va fi de forma: x=(1,1,2,2), adică (()).
Vom retine in variabila ps numărul de paranteze stângi folosite și în variabila pd numărul de parateze drepte folosite. Identificăm următoarele particularităţi şi condiţii:

- S={1,2}
- vectorul soluție: $X=(x_1, x_2, \ldots x_n) \in S^n$
- x[1]=1; x[n]=2
- Fiecare element $X_k \in \{1,2\}$
- valid(k): ps<=n/2 și ps>=pd
- Obţinem soluţia dacă k=n+1 și ps=pd

8. *Generarea partitiilor unei mulțimi.*

Se consideră multimea {1,2,…,n}. Se cer toate partițiile acestei mulțimi. Submulțimile $A_1, A_2, \ldots, A_k$ ale mulțimii A constituie o partiție a acesteia dacă sunt disjuncte între ele (nu au elemente comune) și mulțimea rezultată în urma reuniunii lor este A.

Exemplu:

Pentru A={1,2,3} avem:
{1}, {2,3}
{1,2}, {3}
{1}, {2}, {3}   5 partitii
{1,2,3}
{1,3}, {2}

O partitie a mulțimii {1,2,3,…,n} se poate reprezenta sub forma unui vector x cu n componente *x[i]=k* are semnificatia ca elementul *i* al mulțimii considerate aparține submulțimii k a partiției.

O solutie este x={1,2,3} care reprezintă partiția {1}, {2,3} formată din submulțimile A1={1} și A2={2,3}.

- X[1]=1 semnifica faptul ca 1 ϵ A1
- X[2]=2 semnifica faptul ca 2 ϵ A2
- X[3]=2 semnifica faptul ca 3 ϵ A2

Submultimile unei partitii se numeroteaza cu numere consecutive. Pentru orice *i* din {1,2,…,n} trebuie sa existe un *j* din aceeasi multime astfel incat |*x[i]-x[j]*|<=1. Nu putem avea ca solutie x(1,1,1,3) pentru ca partiția obținută nu are 3 submultimi, insa putem avea x=(1,2,1,3).

- O partitie a unei multimi cu *n* elemente este formată din cel mult n multimi distincte (de ex {1},{2},{3} partitie a lui {1,2,3}) => S={1,2,3,…,n}
- vectorul soluție: $X=(x_1, x_2, \ldots x_n) \in S^n$
- Pentru a evita repetitia partitiilor (de ex. {1}, {2,3} cu {2,3}, {1}) , facem conventia ca x[k] sa ia numai valori din multimea 1,2,3,…,max=maxim(x[1],x[2],…,x[k-1])+1. Deci $X_k \in \{1,2,\ldots,max\}$
- orice valoare x[k] este validă
- Obţinem soluţia dacă k=n+1

```
//partitiile unei multimi
#include <iostream>
#include <fstream>
using namespace std;

int x[50],a[50], n, nrsol;
void citeste()
{ifstream f("fis.in");
 f>>n;
 int i;
 for(i=1;i<=n;i++)f>>a[i];
}
int maxim(int k)
{int i,z=0;
    for(i=1;i<k;i++)
    z=max(x[i],z);
    return z;
}
void scrie()
{ int i,z,j;
  nrsol++;
  cout<<endl<<"-------------------\n";
  cout<<endl<<"Solutia "<<nrsol<<endl;
  cout<<endl;
  z=maxim(n+1);
  for(i=1;i<=z;i++)
  {cout<<"{ ";
   for(j=1;j<=n;j++)
    if(x[j]==i)cout<<a[j]<<" ";
    cout<<"} ";

}}

void back(int k)
{ if(k==n+1)scrie();
  else
   for(int i=1; i<=maxim(k)+1;i++)
     { x[k]=i;
       back(k+1);}
}

int main()
{ citeste();
x[1]=1;
back(2);
return 0;}
```

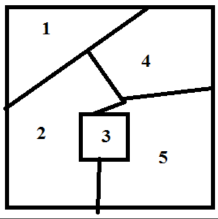
9. Colorarea ţărilor de pe o hartă astfel încât oricare două ţări vecine să aibă culori diferite

Fiind dată o hartă cu *n* țări, se cer toate soluțiile de colorare a hărții, utilizând cel mult 4 culori, astfel încât două țări cu frontiera comună să fie colorate diferit. Este demonstrat faptul că sunt suficiente numai 4 culori pentru ca orice hartă să poată fi colorată.

Harta este furnizată programului cu ajutorul unei matrice A cu n linii și n coloane .

$$A(i,j) = \begin{cases} 1, & \text{daca țara i se învecinează cu țara j;} \\ 0, & \text{in caz contrar.} \end{cases}$$

Matricea A este simetrică. Pentru rezolvarea problemei se utilizeaza vectorul stivă *x*, unde nivelul *k* al acestuia simbolizează țara *k*, iar *x[k]* culoarea atașată țării *k*. Stiva are înălțimea *n* și pe fiecare nivel ia valori între 1 si 4

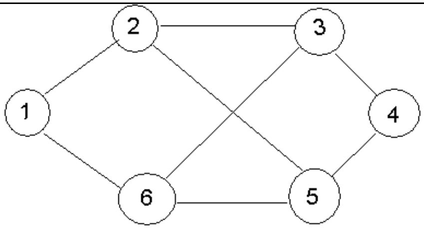
	• S={1,2,3,4} • vectorul soluție: X=(x_1, x_2, ... x_n) ϵ S^n • fiecare element x[k] ϵ {1,2,3,4} • valid(k): A[i][k]=1 și x[i]≠x[k], i=1,2,...,k-1 • Obţinem soluţia dacă k=n+1	0 1 0 1 0 1 0 1 1 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0

```
//colorarea hartilor
#include <iostream>
#include <fstream>
using namespace std;

int x[50],a[50][50];
int n, nrsol;

void citeste()
{ifstream f("fis.in");
 f>>n;
 int i,j;
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
f>>a[i][j];
}

void scrie()
{ int i;
nrsol++;
cout<<endl<<"-------------------\n";
cout<<endl<<"Solutia "<<nrsol<<endl;
for(i=1;i<=n;i++)
  cout<<i<<" ";
cout<<endl;
for(i=1;i<=n;i++)
  cout<<x[i]<<" ";
}

int valid(int k)
{   int i;
    for(i=1;i<k;i++)
      if(a[i][k]==1 &&
      x[i]==x[k])return 0;
    return 1;
}
void back(int k)
{ if(k==n+1)scrie();
  else
   for(int i=1;i<=4;i++)
     { x[k]=i;
       if(valid(k))back(k+1);}
}

int main()
{ citeste(); back(1);
  return 0;
}
```

10. Problema comisului voiajor

Un comis voiajor trebuie să viziteze un număr n de orașe. Inițial, el se află într-unul dintre ele, notat 1. Comisul voiajor dorește să nu treacă de două ori prin același oraș, iar la întoarcere să revină în orașul 1. Cunoscând legăturile existente între orașe, se cere să se afișeze toate variantele de deplasare posibile pe care le poate urma comisul – voiajor.

Exemplu:
In figura de mai jos sunt simbolizate cele 6 orase, precum si drumurile existente intre ele.



Comisul voiajor are urmatoarele posibilitati de parcurgere:

- 1, 2, 3, 4, 5, 6, 1;
- 1, 2, 5, 4, 3, 6, 1;
- 1, 6, 3, 4, 5, 2, 1;
- 1, 6, 5, 4, 3, 2, 1;

Legăturile existente între orașe sunt date în matricea A cu n linii și n coloane. Elementele matricei A pot fi 0 sau 1 (matricea este binară).

$$A(i,j) = \begin{cases} 1, \text{ daca exista drum intre orasele i si j} \\ 0, \text{ in caz contrar.} \end{cases}$$

Se observă ca A[i,j] = A[j,i] oricare ar fi i,j$\in${1, 2, 3, …, n} – matricea este simetrică.

Pentru rezolvarea problemei folosim stiva *X.* La baza stivei (nivelul 1) se incarcă numărul 1, deci x[1]=1. Problema se reduce la a genera toți vectorii X=(x_1,x_2,...,x_n) cu prop:

- `x[1]=1`
- `x[i]`$\in$`{2,3,4,…,n}`
- $x[i] \neq x[j]$, pentru $i \neq j$
- $a[x_1][x_2]=1,\ a[x_2][x_3]=1,\ a[x_{n-1}][x_n]=1,\ a[x_n][1]=1$

Vom utiliza un algoritm asemănător generării permutărilor, pornind de la pasul k=2:

- S={2,3,….,n}
- x[1]=1
- vectorul soluție: X=(x_1, x_2, ... x_n) $\in$ S^n
- x[k] = 1,2,...,n
- valid(k): `A[x[k-1]][x[k]]=1` și $x[i] \neq x[k]$, i=1,2,...,k-1
- obținem soluţia dacă k=n+1 și A`[x[k-1]][x[k]]=1`

G2) probleme în care vectorul soluţie are lungime variabilă şi fiecare element poate să apară de mai multe ori în soluţie;

11. ***Partiţiile unui număr natural***. Fie n>0, natural. Să se scrie un program care să afişeze toate partiţiile unui număr natural n. Numim partiţie a unui număr natural nenul n o mulţime de numere naturale nenule {p1, p2, …, pk} care îndeplinesc condiţia p1+p2+ …+pk = n.

Ex: pt n = 4 programul va afişa:
4 = 1+1+1+1
4 = 1+1+2
4 = 1+3
4 = 2+2
4 = 4

Observaţii: - lungimea vectorului soluţie **X** cel mult n;

- există posibilitatea ca soluţiile să se repete;
- condiţia de final este îndeplinită atunci când suma elementelor vectorului soluţie este n.

Am menţionat mai sus că vom folosi doi parametri, unul pentru poziţia în vectorul soluţie şi un al doilea în care avem sumele parţiale la fiecare moment. Avem determinată o soluţie atunci când valoarea celui de-al doilea parametru este egală cu n.

În această situaţie la fiecare plasare a unei valori în vectorul **X** valoarea celui de al doilea parametru se măreşte cu elementul ce se plasează în vectorul soluţie. Apelul procedurii back din programul principal va fi back(1, 0). Există şi posibilitatea de a apela procedura back din programul principal back(1, n) şi valoarea celui de al doilea parametru se decrementează cu valoarea elementului ce se plasează în vectorul **X**, iar o soluţie avem când acest parametru este zero. Indiferent care modalitate este aleasă acest al doilea parametru ne permite să optimizăm puţin programul în sensul că putem considera nişte condiţii de continuare mai strânse.

```
//generare partitiile unui numar
#include <iostream>
using namespace std;

int x[100], n,k,nrsol;
void citeste()
{cin>>n;}

void scrie(int k)
{ int i; cout<<endl;nrsol++;
  for(i=1;i<k;i++) cout<<x[i]<<" ";}

void back(int k, int sum)
{ if(sum==0)scrie(k);
  else
  for(int i=x[k-1];i<=n && i<=sum;i++)
    //i>=x[k-1]pt a evita repetiiile
     {x[k]=i; back(k+1, sum-i);}
}
int main()
{citeste(); x[0]=1; back(1,n);
cout<<endl<<"nr.solutii="<<nrsol;
return 0;}
```

```
//varianta II
#include <iostream>
using namespace std;

int x[100], n,k,nrsol;
void citeste()
{cin>>n;}

void scrie(int k)
{ int i; cout<<endl;nrsol++;
  for(i=1;i<k;i++) cout<<x[i]<<" ";}

void back(int k, int sum)
{ if( sum==n)scrie(k);
  else
     for(int i=x[k-1];sum+i<=n;i++)
     {x[k]=i; back(k+1, sum+i);}
}

int main()
{citeste(); x[0]=1; back(1,0);
cout<<endl<<"nr.solutii="<<nrsol;
return 0;}
```

12. Plata unei sume cu monede de valori date

Scrieţi un program care să afişeze toate modalităţile prin care se poate plăti o sumă S folosind n bancnote de valori b1<b2<b3<…<bn. Se presupune că avem la dispoziţie oricâte bancnote din fiecare tip. Numerele n şi S, precum şi valorile bancnotelor se citesc de la tastatură, iar modalităţile de plată vor fi scrise în fişierul bani.out.

Indicatie. Se construieşte vectorul y cu numărul maxim de bancnote: y[i]=S/b[i] din fiecare tip

Varianta 1. Vectorul soluție x va avea n componente.

- x[k]є{0,1,2,...,y[k]} are semnificatia: se folosesc x[k] bancnote de tipul b[k]
- valid(k) : suma(x[1]*b[1]+x[2]*b[2]+...+b[k]*x[k])<=S
- solutie: k=n+1 si suma(x[1]*b[1]+x[2]*b[2]+...+b[n]*x[n])=S

```
int valid(int k)
{     return S>=suma(k);}

void back(int k)
{ if(k==n+1){ if(suma(n)==S)scrie();}
        else
        for(int i=0; i<=y[k];i++)
        {
            x[k]=i;
           if(valid(k)) back(k+1);}
}
```

Varianta 2. Vectorul soluție x va avea un numar variabil de componente.

- x[k]є{0,1,2,...,y[k]} are semnificatia: se folosesc x[k] bancnote de tipul b[k]
- valid(k) : suma(x[1]*b[1]+x[2]*b[2]+...+b[k]*x[k])
- solutie: suma(x[1]*b[1]+x[2]*b[2]+...+b[k]*x[k])=S, k<=n+1

```
void back(int k, int sum)
{
    if(k<=n&&sum<S)
    { for(int i=0; i<=y[k] && sum+b[k]*i<=S;i++)
         {x[k]=i; back(k+1, sum+b[k]*i);}
    }
    else if(sum==S)scrie(k);
}
```

13. Submultimi de suma data.

Se da un numar natural nenul s si o multime A={a1,a2,…,an} de numere naturale nenule. Sa se determine toate submultimile lui A cu proprietatea ca suma elementelor acestor submultimi este s.

Exemplu:

Intrucat elementele multimii sunt numere consecutive vom lucra cu indici.

A={8,12,9,5,7,3}, n=6, s=17

x[i]=1,2,…,6 sunt indici care indica pozitia elementului din multime;

x[1]=1 => A[x[1]]=8 ; x[2]=3 => A[x[1]]=9 ;

Indicatie. Problema este un caz particular al problemei anterioare, b[1]=b[2]=b[3]=...=b[n]=1.

<table>
<tr>
<td><pre>int valid(int k)
{   return S>=suma(k);}

void back(int k)
{ if(k==n+1){ if(suma(n)==S)scrie();}
        else
        for(int i=0; i<=1;i++)
        { x[k]=i;
           if(valid(k)) back(k+1);}
}</pre></td>
<td><pre>void back(int k, int sum)
{ if(k<=n&&sum<S)
    { for(int i=0; i<=1;i++)
       if(sum+i*a[k]<=S)
       {x[k]=i; back(k+1, sum+i*a[k]);}
    }
    else if(sum==S)scrie(k);
}</pre></td>
</tr>
</table>

G3) probleme în plan, atunci când spaţiul în care ne deplasăm este un tablou bidimensional

Backtracking generalizat în plan

Metoda Backtracking în plan are câteva modificări:
- stiva conţine mai multe coloane (este dublă, triplă, ...);
- trebuiesc codificate oarecum direcţiile prin numere, litere, elemente, etc.
Problema labirintului se poate rezolva după un algoritm de backtracking generalizat în plan.

14. Problema labirintului

Se dă un labirint sub formă de matrice de n linii şi m coloane. Fiecare element din matrice reprezintă o cameră. Într-una din camerele labirintului se găseşte un om. Se cere să se afle toate soluţiile ca acel om să iasă din labirint, fără să treacă de două ori prin aceeaşi cameră.

OBS:
• O camera poate avea iesire spre alta camera sau in afara labirintului la N, la E, la S sau la V.
• Se poate trece dintr-o cameră în alta, doar dacă între cele două camere există o uşă.
• Prin labirint, putem trece dintr-o cameră în alta, folosind usa, doar mergând în sus N, în jos S, la stânga V sau la dreapta E, nu şi în diagonală.

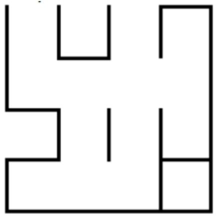
Codificare

• Principiul backtracking generalizat impune codificarea direcţiilor.
• În aceste caz vor fi codificate şi combinaţiile de uși/pereți ai fiecărei camere.
• Asftel, un element al camerei va fi un element al unei matrici cu n linii şi m coloane, având valori de la 0 la 15. În sistemul binar, numerele 0..15 sunt reprezentate ca 0..1111, fiind memorate pe 4 biţi consecutivi.
• Vom lua în considerare toţi cei 4 biţi, astfel numerele vor fi 0000..1110.
• Fiecare din cei 4 biţi reprezintă o direcţie, iar valoarea lui ne spune dacă în acea direcţie a camerei există sau nu o uşă.
• Vom reprezenta numărul astfel: **nr = b1 b2 b3 b4 (b = bit)**
• Asftel, b1 indică direcţia N (sus), b2 indică direcţia E (dreapta), b3 indică direcţia S (jos) iar b4 indică direcţia V (stânga). Valorile unui bit sunt, fireşte, **0** şi **1**.
• **1** înseamnă că în direcţia respectivă *există o uşă*, iar **0** înseamnă că în direcţia respectivă *există un perete, deci pe acolo nu se poate trece*.
• Linia și coloana camerei în care se va deplasa omul din camera curentă se stabilesc astfel:
• dacă direcţia este 1 (sus), linia se va micşora cu 1,
• dacă direcţia este 2 (dreapta), coloana se va mări cu 1,
• dacă direcţia este 3 (jos), linia se va mări cu 1,
• dacă direcţia este 4 (stânga), coloana se va micşora cu 1.

• De exemplu: 1000 - camera aceasta are pereţi în E,S,V, iar în N este uşă spre camera vecină la N. |_| (linia se micșorează cu 1)
• Acest număr este de fapt 8, aşa fiind notat în matricea labirintului. În ceea ce priveşte direcţiile, vom reţine doar coordonatele unde se află omul din labirint, acestea fiind schimbate în funcţie de drumul pe care-l urmează.

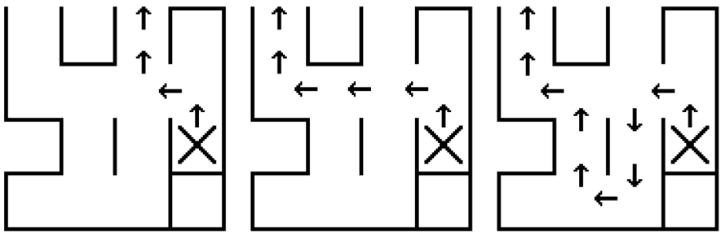
Exemplu:

<table>
<tr>
<td>Să presupunem că avem următoarea matrice:
<code> 9 8 10 2</code>
<code>12 7 15 11</code>
<code> 1 10 10 8</code>
<code> 4 13 9 0</code>

Punctul de pornire din acest labirint este linia 3, coloana 4.</td>
<td>Această matrice corespunde labirintului din desen:
</td>
</tr>
</table>

• Vom avea 3 soluţii de a ieşi din labirint, fără a trece de două ori prin aceeaşi cameră:

- (3,4)-(2,4)-(2,3)-(1,3) -ieşire
- (3,4)-(2,4)-(2,3)-(2,2)-(2,1)-(1,1) -ieşire
- (3,4)-(2,4)-(2,3)-(3,3)-(4,3)-(4,2)-(3,2)-(2,2)-(2,1)-(1,1) –ieşire



• O cameră vizitată se reține prin coordonatele ei lin și col.Pentru a memora coordonatele tuturor camerelor vizitate vom folosi o matrice cu 2 coloane: `d[k][1]=lin; d[k][2]=col;`

Adăugarea coordonatelor unei camere în matricea d se face numai după verificarea existenței acestor coordonate în d, pentru a nu trece de două ori prin acestă cameră. Această verificare se face comparând coordonatele camerei în care suntem cu cele ale camerelor memorate in matricea d.

```
int valid(int i,int j,int k)
{int y;
for(y=1;y<=k;y++)
 if((d[y][1]==i)&&(d[y][2]==j))return 0;
return 1;
}
```

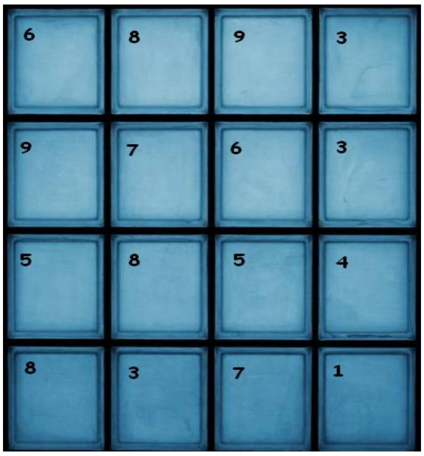
• După metoda Backtracking, trebuiesc găsite toate posibilităţile de a ieşi din labirint.
• S-a ieşit din labirint când linia = coloana = 0, linia = n+1 sau când coloana =m+1.

```
#include <iostream>
#include <fstream>
using namespace std;
int a[20][20],i0,j0,n,m,d[20][3],nr;
ofstream g("a.out");

void citire()
{ int i,j; ifstream f("a.in");
f>>n>>m;
for(i=1;i<=n;i++)
for(j=1;j<=m;j++)  f>>a[i][j];
f>>i0>>j0;
}

void scrie(int k)
{int i;
nr++;
for(i=1;i<=k;i++)
g<<"("<<d[i][1]<<","<<d[i][2]<<")->";
g<<endl;
}

int valid(int i,int j,int k)
{int y;
for(y=1;y<=k;y++)
 if((d[y][1]==i)&&(d[y][2]==j))return 0;
return 1;
}
```

```
void back(int i, int j, int k)
{
 int x=8,ok=0;
   if (valid(i,j,k))
       { d[k+1][1]=i; d[k+1][2]=j;
         if (a[i][j]&x)
            {if(i==1)ok=1;//am iesire
                 else back(i-1,j,k+1);}
         x=x>>1;
         if (a[i][j]&x)
            {if(j==m) ok=1;//am iesire
              else back(i,j+1,k+1);}
         x=x>>1;
         if (a[i][j]&x)
          {if(i==n)ok=1;//am iesire
              else back(i+1,j,k+1);}
         x=x>>1;
         if (a[i][j]&x)
            {if(j==1)ok=1;
              else back(i,j-1,k+1);}
         if(ok)scrie(k+1);
       }
}

int main()
{citire();back(i0,j0,0);
if(nr==0) cout<<"\nNu exista iesire";
else cout<<"\nSunt "<<nr<<" variante";
return 0;}
```

15. Problema Bilei

Se dă un teren sub forma de matrice cu n linii și m coloane. Fiecare element al matricei reprezintă un turn cu o anumită altitudine dată de valoarea reținută de element (număr natural). Pe un astfel de turn, de coordonate (lin,col) se găsește o bilă. Stiind că bila se poate deplasa pe orice turn învecinat aflat la nord, est, sud sau vest, de înălțime strict inferioară turnului pe care se găsește bila, să se găsească toate posibilitățile ca bila să părăsească terenul.



Fie terenul alăturat. Initial, bila se află pe turnul de coordonate (2,2). O posibilitate de iesire din teren este data de drumul:(2,2), (2,3), (3,3), (3,4). In program, altitudinile subteranului vor fi retinute de matricea t.

Initial : (2,2) → Solutii: (2,2) ; (2,3) ; (2,4)
(2,2) ; (2,3) ; (3,3) ; (3,4) ; (2,4)
(2,2) ; (2,3) ; (3,3) ; (3,4)
(2,2) ; (2,3) ; (3,3) ; (3,4) ; (4,4)

Indicație. Problema se poate rezolva folosind algoritmul de la labirint înlocuind testul de intrare într-o camera cu cel de înălțime mai mică.
Nu mai este necesar să testăm dacă bila a ajuns pe un turn deja vizitat, deoarece la fiecare pas, bila se deplasează pe un teren de altitudine strict inferioară.

4 4
6 8 9 3
9 7 6 3
5 8 5 4
8 3 7 1
2 2

```
//bila
#include <iostream>
#include <fstream>
using namespace std;
int a[20][20],i0,j0,n,m,d[20][3],nr;
ofstream g("bila.out");

void citire()
{ int i,j; ifstream f("bila.in");
f>>n>>m;
for(i=1;i<=n;i++)
for(j=1;j<=m;j++)  f>>a[i][j];
f>>i0>>j0;
}

void scrie(int k)
{int i;
nr++;
for(i=1;i<=k;i++)
g<<"("<<d[i][1]<<","<<d[i][2]<<")->";
g<<endl;
}

void back(int i, int j, int k)
{ int ok=0;
  d[k+1][1]=i; d[k+1][2]=j;
  if  (a[i-1][j]<a[i][j])
           {if(i==1)ok=1;//am iesire
            else back(i-1,j,k+1);}
  if  (a[i][j+1]<a[i][j])
           {if(j==m)  ok=1;//am iesire
             else back(i,j+1,k+1);}
  if  (a[i+1][j]<a[i][j])
         {if(i==n)ok=1;//am iesire
             else back(i+1,j,k+1);}
  if  (a[i][j-1]<a[i][j])
           {if(j==1)ok=1;
             else back(i,j-1,k+1);}
  if(ok)scrie(k+1);
}


int main()
{citire(); back(i0,j0,0);
if(nr==0) cout<<"\nNu exista iesire";
else
cout<<"\nSunt  "<<nr<<"  variante  de
iesire";
return 0;
}
```