Metoda Backtracking

Curs 6

Prezentarea generală a metodei

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

- soluţia lor poate fi pusă sub forma unui vector S ={ x_1, x_2,..., x_n }, cu $x_1 \in A_1$, $x_2 \in A_2$,..., $x_n \in A_n$

- mulţimile A_1, A_2 ,...., A_n sunt mulţimi finite, iar elementele lor se consideră că se află într-o relaţie de ordine bine stabilită

- nu se dispune de o altă metodă de rezolvare, mai rapidă

- x_1, x_2 ,..., x_n pot fi la rândul lor vectori

- A_1, A_2 ,..., A_n pot coincide

Prezentarea generală a metodei

- La întâlnirea unei astfel de probleme, dacă nu cunoaştem această tehnică, suntem tentaţi să generăm toate elementele produsului cartezian A_1x A_2 x….x A_n și fiecare element să fie testat dacă este soluţie.

- Rezolvând problema în acest mod, timpul de execuţie este atât de mare, încât poate fi considerat infinit, algoritmul neavând nici o valoare practică.

Prezentarea generală a metodei

- De exemplu, dacă dorim să generăm toate permutările unei mulţimi finite A, nu are rost să generăm produsul cartezian AxAx.....xA, pentru ca apoi, să testăm, pentru fiecare element al acestuia, dacă este sau nu permutare (nu are rost să generăm a_1, a_1, a_1,..., a_1, pentru ca apoi să constatăm că nu am obţinut o permutare, când de la a doua cifră $a_1$ne puteam da seama că cifrele nu sunt distincte).

Prezentarea generală a metodei

Tehnica Backtracking are la bază un principiu extrem de simplu:

- se construieşte soluţia pas cu pas: $x_1, x_2, \ldots, x_n$
- dacă se constată că, pentru o valoare aleasă, nu avem cum să ajungem la soluţie, se renunţă la acea valoare şi se reia căutarea din punctul în care am rămas

Prezentarea generală a metodei

Concret:

- se alege primul element x_1, ce aparţine lui A_1
- presupunând generate elementele $x_1,x_2,\ldots,x_k$, aparţinând mulţimilor A_1, $A_2,\ldots$, A_k, se alege (dacă există) x_{k+1}, primul element disponibil din mulţimea A_{k+1},

apar două posibilităţi:

1) ***Nu s-a găsit un astfel de element***, caz în care se reia căutarea considerând generate elementele $x_1,x_2,\ldots,x_k$, iar aceasta se reia de la următorul element al mulţimii A_k rămas netestat

Prezentarea generală a metodei

2) ***A fost găsit***, caz în care se testează dacă acesta îndeplineşte anumite condiţii de continuare apărând astfel două posibilităţi:

– îndeplineşte, caz în care se testează dacă s-a ajuns la soluţie si apar din nou două posibilităţi:

- s-a ajuns la soluţie, se tipăreşte soluţia si se reia algoritmul considerând generate elementele $x_1,x_2,\ldots,x_k$,

(se caută în continuare, un alt element al mulţimii A_k, rămas netestat)

- nu s-a ajuns la soluţie, caz în care se reia algoritmul considerând generate elementele $x_1,x_2,\ldots,x_{k+1}$, si se caută un prim element $x_{k+2} \in A_k$

Prezentarea generală a metodei

2) ***A fost găsit***, caz în care se testează dacă acesta îndeplineşte anumite condiţii de continuare apărând astfel două posibilităţi:

– nu le îndeplineşte, caz în care se reia algoritmul considerând generate elementele $x_1,x_2,\ldots,x_k$, iar elementul x_{k+1} se caută între elementele mulţimii A_k, rămase netestate

- Algoritmii se termină atunci când nu mai există nici un element $x_1 \in A_1$ netestat.

Observaţie:

Tehnica Backtracking are ca rezultat obţinerea tuturor soluţiilor problemei.

În cazul în care se cere o singură soluţie se poate forţa oprirea, atunci când a fost găsită.

Prezentarea generală a metodei

Am arătat că orice soluţie se generează sub formă de vector.

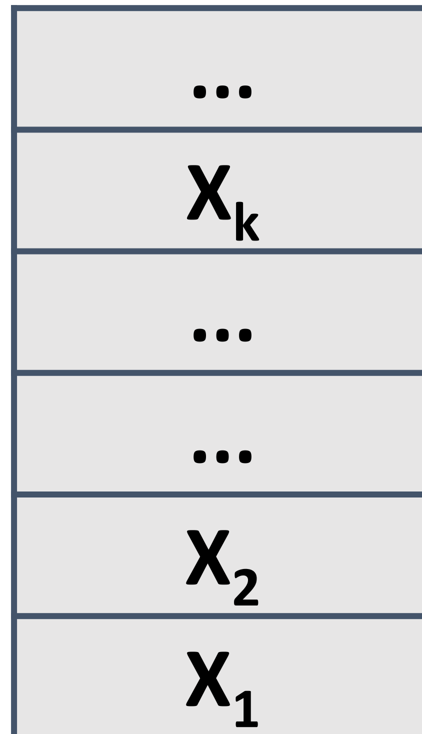
Vom considera că generarea soluţiilor se face într-o stivă.

Metoda backtracking se poate implementa şi recursiv, dar implementarea iterativă foloseşte o stivă, iar principiul de lucrul este cel specific stivei – LIFO (Last In First Out) – Ultimul Intrat Primul Iesit

Prezentarea generală a metodei

Astfel, $x_1 \in A_1$, se va găsi pe primul nivel al stivei, $x_2 \in A_2$ se va găsi pe al doilea nivel al stivei,..., $x_k \in A_k$ se va găsi pe nivelul k al stivei.

În acest fel, stiva (notată **ST**) va arăta astfel:



Prezentarea generală a metodei

- Nivelul k+1 al stivei trebuie iniţializat (pentru a alege, în ordine, elementele mulţimii k+1).

- Iniţializarea trebuie făcută cu *o valoare aflată* (în relaţia de ordine considerată, pentru mulţimea A_{k+1}) *înaintea tuturor valorilor posibile din mulţime.*

Implementarea metodei backtracking

Generarea permutărilor

Enunţ:

Se citeşte un număr natural n. Se cere să se genereze toate permutările mulţimii {1, 2,…,n}.

Caz particular**: dacă n=3, mulţimile $A_1 = A_2 = A_3$ = {1, 2, 3}**

Soluţiile ar fi:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Observaţie:

Se observă, că elementele mulţimilor sunt în ordine strict crescătoare şi consecutive.

Aşadar modul de alegere din mulţime nu este aleatoriu ci se face în funcţie de ordinea elementelor.

Implementarea metodei backtracking

<u>Generarea permutărilor</u>

Trebuie să stabilim unele detalii cu privire la:

1. **vectorul soluţie** – câte componente are, ce conţine fiecare componentă.

2. **mulţimea de valori posibile** pentru fiecare componentă (sunt foarte importante limitele acestei mulţimi).

3. **condiţiile de continuare** (condiţiile ca o valoare x_k să fie acceptată).

4. **condiţia ca ansamblul de valori generat să fie soluţie.**

Implementarea metodei backtracking

Generarea permutărilor

- pentru generarea permutărilor mulţimii {1,2.....n}, orice nivel al stivei va lua valori de la 1 la n.
- Iniţializarea unui nivel (oarecare) se face cu valoarea 0.
- Procedura de iniţializare o vom numi **INIT** şi va avea doi parametri:

k (nivelul care trebuie iniţializat) si ST (stiva).

Implementarea metodei backtracking

Generarea permutărilor

- Găsirea următorului element al mulţimii A_k (element care a fost netestat) se face cu ajutorul procedurii **SUCCESOR** (AS,ST,K).
- Parametrul AS (am succesor) este o variabilă booleană.
- În situaţia în care am găsit elementul, acesta este pus în stivă şi AS ia valoarea **TRUE**, în caz contrar (nu a rămas un element netestat) AS ia valoarea FALSE.

Implementarea metodei backtracking

Generarea permutărilor

- Odată ales un element, trebuie văzut dacă acesta îndeplineşte condiţiile de continuare (altfel spus, dacă elementul este valid).
- Acest test se face cu ajutorul procedurii ***VALID*** (EV,ST,K).
- Parametrul EV (este valid) este o variabilă booleană.
- În situaţia în care elementul îndeplineşte condiţiile de continuare, EV ia valoarea **TRUE**, in caz contrar (nu îndeplineşte condiţiile de continuare) EV ia valoarea **FALSE**.

Implementarea metodei backtracking

Generarea permutărilor

- Testul dacă s-a ajuns sau nu la soluţia finală se face cu ajutorul funcţiei **SOLUTIE**(k)

- O soluţie se tipăreşte cu ajutorul procedurii **TIPAR**.

Implementarea metodei backtracking

Generarea permutărilor: *Schema generală a algoritmului în pseudocod este*:

```
procedura init
begin
    st[k]<-valoare initiala de pornire
end;

procedura succesor
begin
    daca st[k]<valoarea ultimului element din S atunci
    begin
            st[k]<-st[k]+1;
            as<-true
    end
    altfel as<-false;
end;
```

Implementarea metodei backtracking

```
procedura valid
begin
        ev<-true;
```

În continuare se verifică condiţiile ce trebuiesc îndeplinite de elementele aflate în stiva până la nivelul dat k. Condiţiile se stabilesc astfel încât în cazul îndeplinirii lor variabila ev să primească de fiecare dată valoarea false (condiţiile se neagă!!!)

```
end;

functia solutie()
begin
    daca stiva este plina atunci
            solutie<-true
    altfel solutie<-false;
end;

procedura tipar()
begin
    Se parcurge stiva şi se tipăresc valorile aflate acolo;
end;
```

Implementarea metodei backtracking

Programul principal este de fapt algoritmul cu revenire **backtracking**

```
begin
        se citesc date de intrare
        k<-1;
        init
        cat timp k > 0 executa
        begin
                repeta
                        succesor
                        daca as atunci valid
                pana cand (as si ev) sau (not as)
                daca as atunci
                        daca solutie atunci tipar
                        altfel
                        begin
                                k<-k+1;
                                init
                        end
                altfel
                        k<-k-1;
        end;
end
```

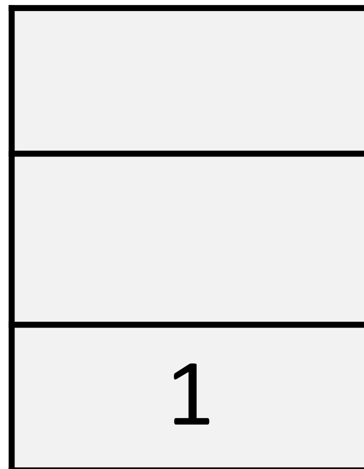
Implementarea metodei backtracking

- Pentru enunţul de la exemplul anterior, generăm, cu ajutorul stivei, soluţiile prin metoda backtracking.

Pentru n=3:

pas 1:

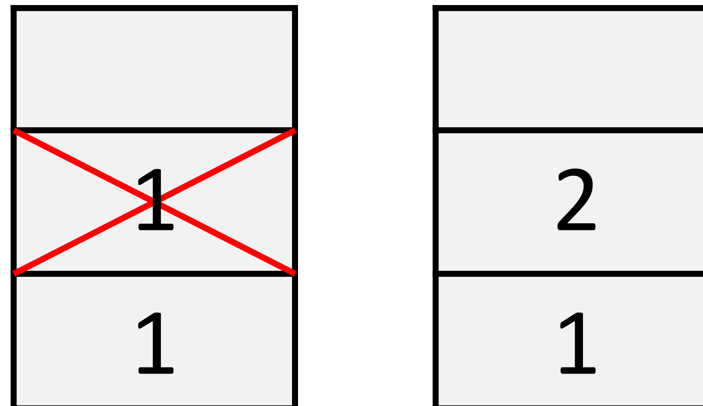
- Se ia primul element din mulţimea A_1 şi se pune pe nivelul 1 al stivei.
- Nu există deocamdată nici o restricţie deoarece există permutări ce încep cu 1.
- Se trece la completarea următorului nivel al stivei.



Implementarea metodei backtracking

pas 2:

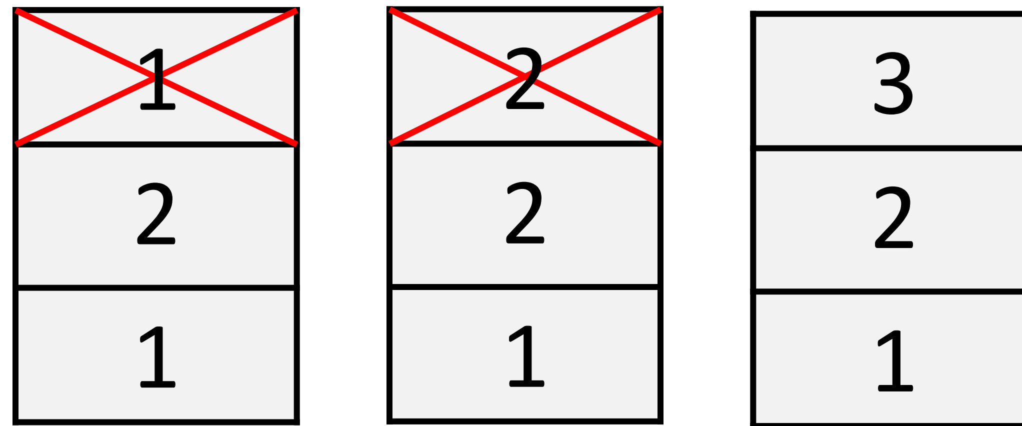
- Se ia primul element din mulţimea A_2 şi se pune pe nivelul 2 al stivei.
- Se observă că valoarea 1 a mai fost introdusă în vectorul soluţie, pe nivelul anterior, şi, în acest caz valoarea 1 nu poate fi ataşată la vectorul soluţie.
- Aşadar, se ia următorul element din mulţimea A_2, adică 2.
- Se constată că poate fi introdusă în vectorul soluţie.
- Se trece la completarea următorului nivel al stivei.



Implementarea metodei backtracking

pas 3:

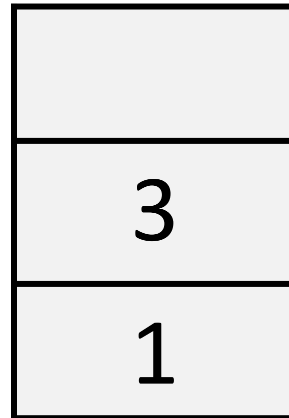
- Se ia primul element din mulţimea A_3 şi se pune pe nivelul 3 al stivei.
- Se observă că valoarea 1 a mai fost introdusă în vectorul soluţie, pe nivelul anterior, şi, în acest caz valoarea 1 nu poate fi ataşată la vectorul soluţie.
- Aşadar, se ia următorul element din mulţimea A_3, adică 2. La fel ca valoarea 1, 2 nu poate fi introdus în stivă.
- Se trece la următorul element din mulţime, 3. Se constată că poate fi introdusă în vectorul soluţie. S-au completat toate nivelele stivei şi deci soluţia este 1 2 3.



Implementarea metodei backtracking

pas 4:

- Pentru nivelul 3 al stivei nu mai există o altă valoare din mulţimea A_3 ce poate fi adăugată.
- Din acest motiv, se revine la nivelul 2 al stivei (înapoi) şi se completează cu următoarea valoare din mulţimea A_2 ce nu a fost utilizată, adică 3.
- Se trece la completarea următorului nivel al stivei.



- Procedeul se repetă până când nu se mai poate introduce o valoare pe nivelul 1 al stivei. În acel moment s-au generat toate soluţiile problemei.

Implementarea metodei backtracking

```
void init(int k,int st[ ])
begin
        st[k] -> 0
end
int succesor(int k,int st[ ])
begin
        daca st[k]<n
        begin
                st[k] -> st[k]+1
                as -> 1 //true
        end
        altfel as -> 0 //false
return as
end
```

Implementarea metodei backtracking

```
int valid(int k,int st[ ])
begin
        ev -> 1
        pentru i de la 1 la k-1
                daca st[i] = st[k]
                        ev -> 0
        return ev
end
int solutie(int k)
begin
        daca k=n  return 1
        altfel return 0
end
```

```
void tipar(void)
begin
        pentru i de la 1 la n
        afiseaza st[i]
end
```

Implementarea metodei backtracking

```
int main(void)
begin
   k=1
   init(k,st)
   cat timp k>0  //while
   begin
         executa //do
         begin
                as=succesor(k,st)
                daca as atunci
                       ev=valid(k,st)
         end //do
         pana cand(!( !as || (as && ev)))

daca as  //tot in cadrul while
        daca solutie(k) atunci tipar()
        altfel
        begin
               k++
               init(k,st)
        end
else  k--

end  //end while

end // end main
```

Implementarea metodei backtracking

```
Dati n = 3
 1 2 3
 1 3 2
 2 1 3
 2 3 1
 3 1 2
 3 2 1


--------------------------------
Process exited after 1.865 seconds
Press any key to continue . . .
```

```
Dati n = 4
 1 2 3 4
 1 2 4 3
 1 3 2 4
 1 3 4 2
 1 4 2 3
 1 4 3 2
 2 1 3 4
 2 1 4 3
 2 3 1 4
 2 3 4 1
 2 4 1 3
 2 4 3 1
 3 1 2 4
 3 1 4 2
 3 2 1 4
 3 2 4 1
 3 4 1 2
 3 4 2 1
 4 1 2 3
 4 1 3 2
 4 2 1 3
 4 2 3 1
 4 3 1 2
 4 3 2 1


--------------------------
Process exited after 2.138
```

Implementarea metodei backtracking

Problema 2:

Fiind dată o tablă de şah, de dimensiune n*n, se cer toate soluţiile de aranjare a n regine, astfel încât să nu se afle două regine pe aceeaşi linie, coloană sau diagonală (reginele să nu se atace reciproc).

Afişarea să se facă astfel:

Exemplu: Pentru n=4 se va afişa:

Soluţia 1:

* R * *

* * * R

R * * *

* * R *

Soluţia 2:

* * R *

R * * *

* * * R

* R * *

Implementarea metodei backtracking

```
Dati numarul de regine = 4
Solutia 1
 *  R  *  *
 *  *  *  R
 R  *  *  *
 *  *  R  *
Solutia 2
 *  *  R  *
 R  *  *  *
 *  *  *  R
 *  R  *  *


--------------------------------
Process exited after 1.365
Press any key to continue .
```

```
Dati numarul de regine = 6
Solutia 1
 *  R  *  *  *  *
 *  *  *  R  *  *
 *  *  *  *  *  R
 R  *  *  *  *  *
 *  *  R  *  *  *
 *  *  *  *  R  *
Solutia 2
 *  *  R  *  *  *
 *  *  *  *  *  R
 *  R  *  *  *  *
 *  *  *  *  R  *
 R  *  *  *  *  *
 *  *  *  R  *  *
Solutia 3
 *  *  *  R  *  *
 R  *  *  *  *  *
 *  *  *  *  R  *
 *  R  *  *  *  *
 *  *  *  *  *  R
 *  *  R  *  *  *
Solutia 4
 *  *  *  *  R  *
 *  *  R  *  *  *
 R  *  *  *  *  *
 *  *  *  *  *  R
 *  *  *  R  *  *
 *  R  *  *  *  *


--------------------------------
Process exited after 1.13 seconds
```

Problema 3

Sa se genereze toate aranjamentele multimii {1, 2, …, n}, cu cate p elemente.

Exemplu:

Pentru n = 5 si p = 3 se vor afisa multimile:

(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1);
(1,2,4), (1,4,2), (2,1,4), (2,4,1), (4,1,2), (4,2,1);
(1,2,5), (1,5,2), (2,1,5), (2,5,1), (5,1,2), (5,2,1);
(1,3,4), (1,4,3), (3,1,4), (3,4,1), (4,1,3), (4,3,1);
(1,3,5), (1,5,3), (3,1,5), (3,5,1), (5,1,3), (5,3,1);
(1,4,5), (1,5,4), (4,1,5), (4,5,1), (5,1,4), (5,4,1),
s.a.m.d.

Implementarea programului pentru generarea aranjamentelor:

```
#include<iostream.h>
int st[20],k,p,as,ev;
void init(int k,int st[ ])
{
  st[k]=0;
}
int succesor(int k,int st[ ])
{
   if ( st[k]<n )
          {
           st[k]=st[k]+1;
           as=1;
           }
       else as=0;
     return as;
}
```

```
int valid(int k,int st[ ])
{
   ev=1;
   for (i=1;i<=k-1;i++)
         if (st[i]==st[k]) ev=0;
   return ev;
}
```

```
int solutie(int k)
{
if ( k==p ) return 1;
        else return 0;
}
void tipar(void)
{
  for(i=1;i<=k;i++)
  cout<<"   "<<st[i];
  cout<<"\n";
 }
```

Sigura modificare fata de algoritmul generarii permutarilor!

```
int main(void)
{
   cout<<“Dati n = “; cin>>n;
   cout<<“Dati p = “; cin>>p;
   k=1; init(k,st);
   while (k>0)
  {
        do{
            as=succesor(k,st); if
            (as) ev=valid(k,st);
         }while (!( !as || (as && ev)));
        if (as)
               if (solutie(k)) tipar();
               else
                {
                   k++;
                   init(k,st);
                }
        else
                   k--;
  }
}
```

Problema 4

Sa se genereze toate combinarile multimii {1, 2, …, n}, cu cate p elemente.

Exemplu:

Pentru n = 5 si p = 3 se vor afisa multimile:

(1,2,3), (1,2,4), (1,2,5), (1,3,4), (1,3,5), (1,4,5), (2,3,4), (2,3,5), (3,4,5).

```
int valid(int k,int st[ ])
{
```

Modificare fata de algoritmul generarii permutarilor:

- Elementele trebuie sa fie in ordine crescatoare

```
}
```

```
int solutie(int k)
{

    Generam p combinari !
    comparam k cu p

}
```

```
Dati n = 5
Dati p = 3
  1  2  3
  1  2  4
  1  2  5
  1  3  4
  1  3  5
  1  4  5
  2  3  4
  2  3  5
  2  4  5
  3  4  5


Terminated with return code
Press any key to continue ..
```

Problema 3

Se considera un numar n natural nenul si se cere sa se afiseze toate descompunerile numarului n in suma de numere naturale.

Exemplu:

Pentru n=5 se vor afisa valorile:

1+1+1+1+1

1+1+1+2; 1+1+2+1; 1+2+1+1; 2+1+1+1

1+2+2; 2+1+2; 2+2+1

1+1+3; 1+3+1; 3+1+1

1+4; 4+1

2+3; 3+2

5

int valid(int k,int st[ ])

- {
  - Trebuie calculata suma elementelor aflate la un moment dat in stiva
  - Daca suma depaseste pe n, atunci ev=0!
  - s+=st[i];
- if(s > n) ev=0;
- return ev;
- }

```
int solutie(int k)
{
```

- Trebuie calculata suma elementelor aflate la un moment dat in stiva
- Daca suma este egala cu n, atunci am gasit o solutie!

```
}
```

Problema 4

Să se descompună un număr natural n, în toate modurile posibile, ca sumă de p numere naturale (p<=n).

Exemplu:

Pentru n=5 si p=3 se obtin solutiile:

1+1+3; 1+3+1; 3+1+1;

1+2+2; 2+1+2; 2+2+1;

```
int solutie(int k)
{
        ...
        s+=st[i]
   if( s == n && k == p ) ...
}
```

- Trebuie calculata suma elementelor aflate la un moment dat in stiva
- Daca suma este egala cu n, atunci am gasit o solutie!
- In plus trebuie sa avem fix p elemente in stiva!

```
Dati n = 5
Dati p = 3
   1   1   3
   1   2   2
   1   3   1
   2   1   2
   2   2   1
   3   1   1


Terminated with return code
Press any key to continue .
```

Problema 5

Problema colorării hărților

Fiind dată o hartă cu n țări se cere o soluție de colorare a hărții utilizând cel mult 4 culori, astfel încât două cu frontiera comună să fie colorate diferit.

O soluție este:

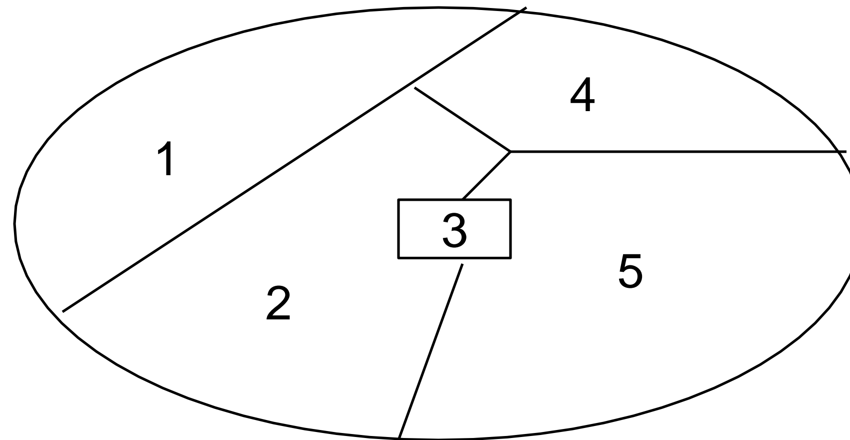
Țara 1 – culoarea 1

Țara 2 – culoarea 2

Țara 3 – culoarea 1

Țara 4 – culoarea 3

Țara 5 – culoarea 4



Pentru a specifica harta utilizam o matrice patratica cu valori binare:

$$A(i,j) = \begin{cases} 1, & \text{daca țara i are frontieră comună cu țara j} \\ 0, & \text{altfel} \end{cases}$$

- Se va utiliza stiva st, unde nivelul k al stivei simbolizează ţara k, iar st[k] culoarea ataşată ţării k.
- Stiva are înălţimea n şi pe fiecare nivel ia valori intre 1 şi 4, adică numărul culorilor este maxim 4.
- Condiţia ca două ţări vecine sa aibă aceeaşi culoare este:

(st[k]==st[i]) && (a[i][k])==1)

```
#include<iostream.h>
#include<stdlib.h>
int st[20],k,as,ev,n,i,j,a[20][20];
void init(int k,int st[ ])
{
  st[k]=0;
}
int succesor(int k,int st[ ])
{
   if ( st[k]<4 )
          {
           st[k]=st[k]+1;
           as=1;
           }
       else as=0;
     return as;
}
```

```
int valid(int k,int st[ ])
{
   ev=1;
   for(i=1;i<=k-1;i++)
        if(st[k] == st[i] && a[i][k] == 1) ev=0;
   return ev;
}
```

```
    int solutie(int k)
{
  if( k == n ) return 1;
        else return 0;
}
void tipar(void)
{
 for(i=1;i<=k;i++)
   cout<<"tara "<<i<<" are culoarea "<<st[i]<<"\n";
 exit(1);
 }
```

```
Dati numarul de tari = 5
a[1][2]=1
a[1][3]=0
a[1][4]=1
a[1][5]=0
a[2][3]=1
a[2][4]=1
a[2][5]=1
a[3][4]=0
a[3][5]=1
a[4][5]=1
tara 1 are culoarea 1
tara 2 are culoarea 2
tara 3 are culoarea 1
tara 4 are culoarea 3
tara 5 are culoarea 4


Terminated with return code
Press any key to continue .
```

Backtracking recursiv

Functiile folosite sunt in general aceleasi, cu doua mici exceptii:

1. In functia SOLUTIE conditia este n+1;
2. rutina backtracking se transforma in functie, care se apeleaza prin BACK(1)

Principiul de functionare al functiei BACK, corespunzator unui nivel k este urmatorul:

- in situatia in care avem o solutie, o tiparim si revenim pe nivelul anterior
- in caz contrar se initializeaza nivelul si se cauta un succesor
- cand am gasit unul verificam daca este valid; functia se autoapeleaza pentru (k+1), in caz contrar urmand a se continua cautarea succesorului;
- daca nu avem succesor, se trece pe nivel inferior (k-1) prin iesirea din functia BACK

Implementarea algoritmului pentru generarea permutarilor – varianta recursiva:

```
#include<iostream.h>
int st[20],k,p,as,ev,n;
void init(int k,int st[ ])
{
  st[k]=0;
}
int succesor(int k,int st[ ])
{
   if ( st[k]<n )
          {
           st[k]=st[k]+1;
           as=1;
           }
     else as=0;
     return as;
}
```

```
int valid(int k,int st[ ])
{
   ev=1;
   for (int i=1;i<=k-1;i++)
              if (st[i]==st[k]) ev=0;
   return ev;
}
int solutie(int k)
{
     if ( k==n+1 ) return 1;
     else return 0;
}
void tipar(void)
{
  for(int i=1;i<=n;i++)
     cout<<"  "<<st[i];
  cout<<"\n";
 }
```

```
void back(int k)
{
    if(solutie(k)) tipar();
    else{
            init(k,st);
            while(succesor(k,st))
            {
                ev=valid(k,st);
                if(ev) back(k+1);
            }
        }
}
int main(void)
{
  cout<<"Dati n = "; cin>>n;
  back(1);
}
```