8. Metoda de programare *Backtracking*

8.1. Prezentare generală

Imaginaţi-vă că astăzi este ziua vostră şi aveţi invitaţi. Aranjaţi o masă frumoasă, apoi vă gândiţi cum să vă aşezaţi invitaţii la masă. Aţi vrea să ştiţi toate posibilităţile de aşezare a invitaţilor la masă, dar realizaţi în acelaşi timp că trebuie să ţineţi seama şi de preferinţele lor. Printre invitaţi există anumite simpatii dar şi unele antipatii, de care doriţi neapărat să ţineţi seama, pentru ca petrecerea să fie o bucurie pentru toţi.

Să ne gândim cum procedaţi pentru a identifica toate posibilităţile de a plasa invitaţii la masă. Începeţi prin a scrie nişte cartonaşe cu numele invitaţilor.

Alegeţi un invitat.

Pentru a-l alege pe al doilea, selectaţi din mulţimea cartonaşelor rămase un alt invitat. Dacă ştiţi că cele două persoane nu se agreează, renuntaţi la cartonaşul lui şi alegeţi altul şi aşa mai departe.

Se poate întâmpla ca la un moment dat, când vreţi să alegeţi cartonaşul unui invitat să constataţi că nici unul dintre invitaţii rămaşi nu se potriveşte cu ultima persoană slectată până acum. Cum procedaţi?

Schimbaţi ultimul invitat plasat cu un invitat dintre cei rămaşi şi încercaţi din nou, dacă nici aşa nu reuşiti, schimbaţi penultimul invitat cu alcineva şi încercaţi din nou şi aşa mai departe până când reuşiţi să plasati toţi invitaţii. Înseamnă că aţi obţinut o soluţie.

Pentru a obţine toate celelalte soluţii, nu vă rămâne decât să o luaţi de la început. Aveţi cam mult de muncit, iar dacă numărul invitaţilor este mare...operaţiunea devine foarte anevoioasă. Iată de ce aveţi nevoie de un calculator şi cunoştinţe de programare .

Backtracking este o metodă de parcurgere sistematică a spaţiului soluţiilor posibile al unei probleme.

Este o metodă generală de programare, şi poate fi adaptă pentru orice problemă pentru care dorim să obţinem toate soluţiile posibile, sau să selectăm o soluţie optimă, din mulţimea soluţiilor posibile.

Backtracking este însă şi cea mai costisitoare metodă din punct de vedere al timpului de execuţie.

În general vom modela soluţia problemei ca un vector $\mathbf{v=(v_1,v_2,v_3,...,v_n)}$ în care fiecare element $\mathbf{v_k}$ aparţine unei mulţimi finite şi ordonate $\mathbf{S_k}$, **cu k=1,n**. În anumite cazuri particulare, mulţimile S_1 ,S_2, S_3,...,S_n pot fi identice . Procedăm astfel:

1. La fiecare pas k pornim de la o soluţie parţială $v=(v_1,v_2,v_3,...,v_{k-1})$ determinată până în acel moment şi încercăm să extindem această soluţie adăugând un nou element la sfârşitul vectorului.
2. Căutăm în mulţimea S_k , un nou element.
3. Dacă există un element neselectat încă, verificăm dacă acest element îndeplineşte condiţiile impuse de problemă, numite **condiţii de continuare.**
4. Dacă sunt respectate condiţiile de continuare, adăugăm elementul soluţiei parţiale.
5. Verificăm dacă am obţinut o soluţie completă.

- dacă am obţinut o soluţie completă o afişăm şi se reia algoritmul de la pasul 1.
- dacă nu am obţinut o soluţie, k <----- k+1 si se reia algoritmul de la pasul 1.

6. Dacă nu sunt respectate condiţiile de continuare se reia algoritmul de la pasul 2.
7. Dacă nu mai există nici un element neverificat în mulţimea S_k înseamnă că nu mai avem nici o posibilitate din acest moment, să construim soluţia finală aşa că trebuie să modificăm alegerile făcute în prealabil, astfel k <----- k-1 şi se reia problema de la pasul 1.

Revenirea în caz de insucces sau pentru generarea tuturor soluţiilor problemei, a condus la denumirea de ”backtracking” a metodei, traducerea aproximativă ar fi “revenire în urmă”.

Forma generală a unei funcţii backtracking

Implementarea recursivă a algoritmului furnizat de metoda backtracking, este mai naturală şi deci mai uşoară. Segmentul de stivă pus la dispoziţie prin apelul funcţiei este gestionat în mod automat de sistem. Revenirea la pasul precedent se realizează în mod natural prin închiderea nivelului de stivă.

```
void BK(int k)                          //k-poziţia din vector care se completează
{int i;
 for (i=1; i<=nr_elemente_Sk; i++)      //parcurge elementele mulţimii Sk
  { v[k]=i;                             //selectează un element din mulţime
    if (validare(k)==1)                 //validează condiţiile de continuare ale problemei
     {  if (solutie(k)==1)              //verifică dacă s-a obţinut o soluţie
          afisare(k);                   //afişează soluţia
       else
          BK(k+1);                      //reapelează functia pentru poziţia k+1
     }
  }                                     //dacă nu mai există nici un element neselectat în mulţimea Sk,
}                                       //se închide nivelul de stivă şi astfel se revine pe poziţia k-1 a
                                        //vectorului
//execuţia funcţiei se încheie, după ce s-au închis toate nivelurile stivei, înseamnă că în vectorul v nu
mai poate fi selectat nici un elemente din multimile Sk
```

8.2. Exemple de implementare a metodei:

1. **PERMUTĂRI**

Să se genereze toate permutările primelor **n** numere naturale.

Vom genera pe rând soluţiile problemei în vectorul $v=(v_1,v_2,v_3,...,v_n)$, unde $v_k \in S_k$.
Să facem următoarele observaţii:

1. Pentru această problemă toate mulţimile S_k sunt identice, $S_k=\{1,2,3,....,n\}$. La pasul **k** selectăm un element din mulţimea S_k.
2. Întrucât în cadrul unei permutări **elementele nu au voie să se repete** această condiţie reprezentă condiţia de continuare a problemei.
3. Obţinem o soluţie în momentul în care completăm vectorul cu **n** elemente.

Exemplu pentru n=3 $S_1= S_2= S_3=\{1,2,3\}$
(1,2,3) (1,3,2) (2,1,3) (2,3,1) (3,1,2) (3,2,1)

v **k=1**

1		
k 1	2	3

v **k=2**

1	1	
k 1	2	3

element incorect (se repetă)

v **k=2**

1	2	
k 1	2	3

v **k=3**

1	2	1
k 1	2	3

element incorect (se repetă)

v **k=3**

1	2	2
k 1	2	3

element incorect (se repetă)

v **k=3**

1	**2**	**3**
k 1	2	3

k=3 s-a obţinut o soluţie

se închide ultimul nivel de stivă pentru că nu mai sunt elemente în ultima mulţime

v **k=2**

1	3	
k 1	2	3

v **k=3**

1	3	1
k 1	2	3

element incorect (se repetă)

v

1	**3**	**2**
k 1	2	3

k=3

s-a obţinut o soluţie

v **k=3**

1	3	3
k 1	2	3

element incorect (se repetă)

v **k=2**

1	3	
k 1	2	3

v **k=1**

2		
k 1	2	3

se repetă aceiaşi paşi construindu-se soluţiile următoare

STIVA

i=1

STIVA

i=**1**
i=1

STIVA

i=1,**2**
i=**1**

STIVA

i=1
i=1,**2**
i=**1**

STIVA

i=1,**2**
i=1,**2**
i=**1**

STIVA

i=1,2,**3**
i=1,**2**
i=**1**

STIVA

i=1,2,**3**
i=**1**

STIVA

i=1
i=1,2,**3**
i=**1**

STIVA

i=1,**2**
i=1,2,**3**
i=**1**

STIVA

i=1,2,**3**
i=1,2,**3**
i=**1**

STIVA

i=1,2,**3**
i=**1**

STIVA

i=1,**2**

```
#include <iostream.h>   //              PERMUTĂRI
const MAX=20;
int n,v[MAX] ;          //n-nr. de elemente, v[20]-vectorul în care construim soluţia
int valid(int k);
int solutie(int k);
void afisare(int k);
void BK(int k);

int main()
{cout<<"n= ";cin>>n;   //se citeşte n
 BK(1);
 return 0;              //apelăm funcţia BK pentru completarea poziţiei 1din vectorul v
}
void BK(int k)
{int i;                 //i-elementul selectat din multimea Sk, trebuie sa fie variabilă locală, pentru
                        // a se memora pe stivă
 for (i=1;i<=n;i++)     //parcurgem elementele mulţimii Sk
  {v[k]=i;              //selectăm un element din mulţimea Sk
   if (valid(k))        //verificăm dacă eelementul ales îndeplineşte condiiile de continuare
    {if (solutie(k))    //verificăm dacă am obţinut o soluţie
      afisare(k);       //se afişează soluţia obţinută
     else
      BK(k+1);          //reapemăm funcţia pentru poziţia k+1 din vectorul v
    }
 }
}
int valid(int k)        //verificăm condiţiile de continuare
{int i;
 for (i=1;i<=k-1;i++)   //comparăm fiecare element din vectorul v cu ultimul element selectat
  if (v[i]==v[k])       //deoarece într-o  permutare elementele nu au voie să se repete,
   return 0;            //returnăm 0 în cazul în care elementul selectat, a mai fost selectat o dată
 return 1;              //returnăm 1 în cazul în care elementul nu mai apare în vector
}
int solutie(int k)      //verificăm dacă am obţinut o soluţie
{if (k==n)              //am obţinut o permutare, dacă am reuşit să depunem în vector n elemente
   return 1;
 return 0;
}
void afisare(int k)     //afişează conţinutul vectorului v
{int i;
 for (i=1;i<=k;i++)
  cout<<v[i]<<" ";
 cout<<endl;
```

Problema generării permutărilor, este cea mai reprezentativă pentru metoda backtracking, ea conţine toate elementele specifice metodei.

Probleme similare, care solicită determinarea tuturor soluţiilor posibile, necesită doar adaptarea acestui algoritm modificând fie modalitatea de selecţie a elementelor din mulţimea S_k, fie condiţiile de continuare fie momentul obţinerii unei soluţii.

2. PRODUS CARTEZIAN

Se dau **n** mulţimi, ca cele de mai jos:
$S_1=\{1,2,3,...,w_1\}$
$S_2=\{1,2,3,...,w_2\}$
.....
$S_n=\{1,2,3,...,w_n\}$
Se cere să se gnereze produsul lor cartezian.

Exemplu:
pemtru n=3 şi următoarele mulţimi
$S_1=\{1,2\}$ $w_1=2$
$S_2=\{1,2,3\}$ $w_2=3$
$S_3=\{1,2\}$ $w_3=2$
produsul cartezian este:
$S_1 xS_2 xS_3 =\{ (1,1,1),(1,1,2),(1,2,1),(1,2,2),(1,3,1),(1,3,2),$
$(2,2,1),(2,1,2),(2,2,1),(2,2,2),(2,3,1),(2,3,2)\}$
Prin urmare o soluţie este un şir de n elemente, fiecare element $i \epsilon S_i$, cu i=1,n
Să analizăm exemplul de mai sus:

1. La pasul **k** selectăm un element din mulţimea $\mathbf{S_k =\{1,2,3,...,w_k\}}$.
2. Elementele unei soluţii a produsului cartezian, pot să se repete, pot fi în orice ordine, iar valori străine nu pot apare, deoarece le selectăm doar dintre elementele mulţimii S_k . Prin urmare **nu trebuie să impunem condiţii de continuare.**
3. Obţinem o soluţie în momentul în care am completat **n** elemente în vectorul v.

Vom memora numărul de elemente al fiecărei mulţimi S_k , într-un vector **w**. Soluţiile le vom construi pe rând în vectorul **v**.

```
#include <iostream.h>          //    PRODUS CARTEZIAN
#include <fstream.h>
const MAX=20;
int n,v[MAX],w[MAX];           //n-nr. de mulţimi, v-vectorul soluţie, w-conţine nr. de elemente di
                               //fiecare mulţime Sk
void BK(int k);
void citire();
void afisare(int k);
int solutie(int k);
void main()
{citire();                     //citire date
 BK(1);                        //apelăm funcţia BK pentru selectarea primului element în v
}
void BK(int k)                 //funcţia becktreacking
{int i;
 for (i=1;i<=w[k];i++)         //parcurgem mulţimea Sk ={1,2,3,...,wk}
  {v[k]=i;                     //selectăm elementul i din mulţimea Sk
                               //nu avem funcţie de validare- nu avem condiţii de continuare
   if (solutie(k))             //verificăm dacă am obţinut o soluţie
     afisare(k);               //afişăm soluţia
   else
    BK(k+1);                   //reapelăm funcia BK pentru completarea poziţiei următoare în
                               // vectorul v
  }                            //se închide un nivel de stivă si astfel se ajunge la poziţia k-1 în v
}
```

```
void citire()                    //citirea datelor
{int i;
 ifstream f("prod.in");
 f>>n;                           //se citeşte numărul de mulţimi
 for(i=1;i<=n;i++)
    f>>w[i];                     //se citeşte numărul de elemente al fiecărei mulţimi
 f.close();
}
int solutie(int k)               //funcţia soluţie determină momentul în care se ajunge la o soluţie
{if (k==n)                       //obţinem o soluţie dacă am dpus în vectorul v, n elemente
   return 1;
  return 0;
}
void afisare(int k)              //afuşează o soluţie
{int i;
 for (i=1;i<=k;i++)
    cout<<v[i]<<" ";
 cout<<endl;
}
```

3. ARANJAMENTE

Se citesc n şi p numere naturale cu p<=n. Sa se genereze toate aranjamentle de n elemente luate câte p.
Exemplu pentru n=3, p=2
(1,2), (1,3), (2,1), (2,3), (3,1), (3,2)

Vom genera pe rând soluţiile problemei în vectorul $\mathbf{v=(v_1,v_2,v_3,...,v_n)}$, unde $\mathbf{v_k \in S_k}$.
Să facem următoarele observaţii:

1. pentru această problemă toate mulţimile $\mathbf{S_k}$ sunt identice, $\mathbf{S_k=\{1,2,3,....,n\}}$.
2. la pasul **k** selectăm un element din mulţimea $\mathbf{S_k}$.
   Întrucât în cadrul unei aranjări, **elementele nu au voie să se repete** această condiţie reprezentă condiţia de continuare a problemei.
3. Oţinem o soluţie în momentul în care completăm vectorul cu **p** elemente.

Să observăm că problema generării aranjamentelor, nu diferă prea mult de problema generării permutărilor. Singura deosebire pe care o sesizăm este aceea că obţinem o soluţie în momentul în care am plasat în vector **p** elemente.
Prin urmare, în cadrul programului pentru generarea permutărilor trebuie sa modificăm o singură funcţie şi anume funcţia *soluţie,* astfel:

```
int solutie(int k)               //verificăm dacă am obţinut o soluţie
{if (k==p)                       //am obţinut o aranjare, dacă am reuşit să depunem în vector p elemente
   return 1;
 return 0;
}
```

4. COMBINĂRI

Se citesc n şi p numere naturale cu p<=n. Să se genereze toate combinările de n elemente luate câte p.
Exemplu pentru n=3, p=2. obţinem (1,2), (1,3), (2,3)

Vom genera pe rând soluţiile problemei în vectorul **$v=(v_1,v_2,v_3,...,v_n)$**, unde **$v_k \varepsilon S_k$**.
Să facem următoarele observaţii:

1. Pentru această problemă toate mulţimile **S_k** sunt identice, **$S_k=\{1,2,3,....,n\}$**.
   La pasul **k** selectăm un element din mulţimea **S_k**.
2. În cadrul unei combinări elementele nu au voie să se repete.
   Să mai observăm şi faptul că dacă la un moment dat am generat de exemplu soluţia (1,2), combinarea (2,1) nu mai poate fi luată în considerare, ea nu mai reprezintă o soluţie. Din acest motiv vom considera că elementele vectorului reprezintă o soluţie, numai dacă se află în ordine strict crescătoare.
   Acestea reprezintă condiţiile de continuare ale problemei.
3. Oţinem o soluţie în momentul în care vectorul conţine **p** elemente.

Putem genera toate elementele unei combinări, parcurgând mulţimea {1,2,3,...,n}, apoi să verificăm condiţiile de continuare aşa cum am procedat în cazul permutărilor.

Putem însă îmbunătăţii timpul de execuţie, selectând din mulţimea {1,2,3,...,n}, la pasul k un element care este în mod obligatoriu mai mare decăt elementul v[k-1], adică i=v[k-1]+1.

Ce se întîmplă însă cu primul element plasat în vectorul v?
Acest element a fost plasat pe poziţia 1, iar vectorul v deţine şi elementul v[0] în mod implicit în C++. v[0]=0, deoarece vectorul v este o variabilă globală şi se iniţializează automat elementele lui cu 0.

În acest fel, impunând aceste restricţii încă din momentul selecţiei unui element, condiţiile de continuare vor fi respectate şi nu mai avem nevoie de funcţia valid.

Algoritmul a fost substanţial îmbunătăţit, deoarece nu selectăm toate elementele mulţimii şi nu verificăm toate condiţiile de continuare, pentru fiecare element al mulţimii.

```
#include <iostream.h>                    //     COMBINĂRI
const MAX=20;
int n,p,v[MAX] ;
int solutie(int k);
void afisare(int k);
void BK(int k);
void main()
{cout<<"n= ";cin>>n;    cout<<"p= ";cin>>p;
 BK(1);
}
void BK(int k)
{int i;
 for (i=v[k-1]+1;i<=n;i++)  //la pasul k selectăm din mulţime un element mai mare decât elementul
  {v[k]=i;                  //de pe poziţia k-1
   if (solutie(k))          //nu este necesar să verificam condiţiile de continuare, ele sunt respectate
     afisare(k);            //datorită modului în care am selectat elementele.
   else
     BK(k+1);
  }
 }
int solutie(int k)
{if (k==p)  return 1;
 return 0;}
void afisare(int k)
{int i;
 for (i=1;i<=k;i++)  cout<<v[i]<<" ";
cout<<endl;
}
```

4. **SUBMULŢIMI**

Să se genereze toate submulţimile mulţimii S={1,2,3, ... ,n}.

Exemplu: pentru n=3, S={1,2,3}, submulţimile sunt următoarele:
Φ-mulţimea vidă, {1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}

Să observăm că pentru a obţine toate submulţimile unei mulţimi, este suficient să generăm pe rând $C_n^1, C_n^2, ..., C_n^{n-1}$, pentru mulţimea S={1,2,3, ... ,n}, la care trebuie să adăugăm mulţimea vidă şi mulţimea S.

În aceste condiţii, este suficient să modificăm doar funcţia principală pentru a genera toate submulţimile şi afişarea datelor ca mulţimi de elemente. Funţiile *BK* şi *soluţie* generează în realitate C_n^p elemente.

```
#include <iostream.h>              //        SUBMULŢIMI 1
const MAX=20;
int n,p,v[MAX] ;
int solutie(int k);
void afisare(int k);
void BK(int k);
void main()
{int i;
 cout<<"n= ";cin>>n;
 cout<<"mulimea vida"<<endl;
 for(p=1;p<=n-1;p++)              //generăm C_n^p elemente
    BK(1);
cout<<"{";
 for(i=1;i<n;i++)
    cout<<i<<", ";
 cout<<n<<"}";
}
void BK(int k)
{int i;
 for (i=v[k-1]+1;i<=n;i++)
  {v[k]=i;
   if (solutie(k))
     afisare(k);
   else
     BK(k+1);
  }
}
int solutie(int k)
{if (k==p)
   return 1;
 return 0;
}
void afisare(int k)
{ cout<<"{ ";
 for (int i=1;i<k;i++)   cout<<v[i]<<", ";
 cout<<v[k]<<" }"<<endl;
}
```

Putem construi un algoritm mai eficient pentru generarea tuturor submulţimilor unei mulţimi.

De exemplu dacă genetăm mulţimile în următoarea ordine:
mulţimea vidă, {1}, {1,2}, {1,2,3}, {2}, {2,3}, {3}

```
#include <iostream.h>              //SUBMULŢIMI 2
const MAX=20;
int n,p,v[MAX] ;
int valid(int k);
int solutie(int k);
void afisare(int k);
void BK(int k);

void main()
{int i;
 cout<<"n= ";cin>>n;
 cout<<"mulimea vida"<<endl;
   BK(1);
}

void BK(int k)
{int i;
 for (i=v[k-1]+1;i<=n;i++)
  {v[k]=i;
    afisare(k);
    BK(k+1);
  }
}
```

5. **Problema celor n dame**

Fiind dată o tablă de şah de dimensiune nxn, se cere să se aranjeze cele n dame în toate modurile posibile pe tabla de şah, astfel încât să nu se afle pe aceeaşi linie, coloană, sau diagonală (damele să nu se atace).

Exemplu pentru n=4 o soluţie este:

	D		
			D
D			
		D	

Observăm că o damă va fi plasată întotdeauna singură pe o linie. Acest lucru ne permite să memorăm fiecare soluţie într-un vector **v**, considerând că o căsută **k** a vectorului reprezintă **linia k** iar conţinutul ei, adică **v[k]** va conţine **numărul coloanei** în care vom plasa regina.

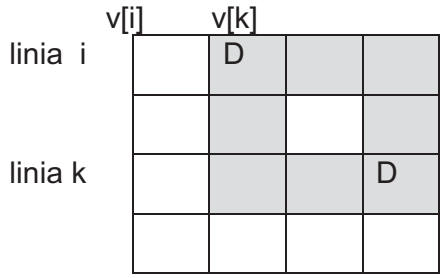
Pentru exemplul de mai sus, vectorul **v** va avea următorul conţinut:

2	4	1	3
1	2	3	4

Să facem următoarele observaţii:

1. Pentru această problemă toate mulţimile S_k sunt identice, $S_k=\{1,2,3,....,n\}$ şi reprezintă numărul coloanelor tablei de şah. Indicele k reprezintă numărul liniei tablei de şah.
   Prin urmare, la pasul **k** selectăm un element din mulţimea S_k.

2. Condiţiile de continuare ale problemei :
   a) Damele să nu fie pe aceeaşi linie - această condiţie este îndeplinită prin modul în care am organizat memorarea datelor şi anume într-o căsuţă a vectorului putem trece un singur număr de coloană.
   b) Damele să nu fie pe aceeaşi coloană – adică v[k]≠v[i], cu 1<=i<=k-1.
   c) Damele să nu fie pe aceeaşi diagonală. Dacă două dame se gasesc pe aceeaşi diagonală înseamnă că cele două distanţe măsutaţe pe linie respectiv pe coloană, dintre cele două dame sunt egale. Prin urmare condiţia ca damele să nu fie pe aceeaşi diagonală este: |v[k]-v[i]| ≠k-i , cu 1<=i<=k-1.



3. Obţinem o soluţie dacă am reuşit să plasăm toate cele n dame, adică k=n.

Să urmărim modul în care se completează vectorul soluţie **v** şi o reprezentare grafică a ceea ce înseamnă valorile vectorului **v** pe tabla de şah.

Reprezentarea descrie completarea vectorului până la obţinerea primei soluţii, pentru n=4.

Algoritmul de backtracking continuă în aceeaşi manieră, până la generarea tuturor soluţiilor.

v **k=1**

1			

v **k=2**

1	1		

v **k=2**

1	2		

k=2

1	3		

pe linia 3 nu mai putem plasa nici o damă, selectăm altă valoare pe linia 2

v **k=2**

1	4		

v **k=3**

1	4	1	

v **k=3**

1	4	2	

-pe linia 4 nu putem plasa nici o damă
-pe linia 3 nici o altă coloană nu este corectă
-pe linia 2 nu mai există nici o poziţie disponibilă
-se revine la linia 1

v **k=1**

2			

v **k=2**

2	**1**		

v **k=2**

2	2		

v **k=2**

2	3		

v **k=2**

2	4		

v **k=3**

2	4	1	

v **k=4**

2	4	1	1

v **k=4**

2	4	1	2

v **k=4**

2	4	1	3

Algoritmul continuă pentru a genera toate soluţiile.

D			

D			
		D	

D			
			D

D			
			D
	D		

	D		

	D		
			D

	D		
			D
D			

	D		
			D
D			
		D	

Am obţinut prima soluţie !

```
#include <iostream.h>              //          DAME
#include <math.h>
#define MAX 20

int n,v[MAX],sol;

int valid(int k);
int solutie(int k);
void afisare();
void BK(int k);

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

void BK(int k)
{int i;
 for (i=1;i<=n;i++)
  {v[k]=i;
   if (valid(k)==1)
    {if (solutie(k)==1)
       sfisare();
     else
       BK(k+1);
    }
  }
}

int valid(int k)
{int i;
 for (i=1;i<=k-1;i++)
   if ((v[i]==v[k])||(abs(v[k]-v[i])==(k-i)))
     return 0;
 return 1;}

int solutie(int k)
{if (k==n)
  return 1;
return 0;}

void afisare()                          //afisam solutiile sub forma unei matrice
{int i,j,x;
 sol++; cout<<"\n Solutia: "<<sol<<'\n';
 for (i=1;i<=n;i++)
  {for (j=1;j<=n;j++)
     if (v[i]==j)     cout<<"D ";
     else       cout<<"_ ";
   cout<<'\n';
  }
}
```

6. Plata unei sume cu monede de valori date.

Fiind dată o sumă S şi n monede de valori date, să se determine toate modalitătile de plată a sumei S cu aceste monede. Considerăm că sunt monede suficiente din fiecare tip.

```
#include <iostream.h>                // PLATA SUMEI
#include <fstream.h>
#define MAX 20
int n=0,x,v[MAX],w[MAX],z[MAX],S,Suma,sol;
//v-vectorul soluţie,w-valoarea monedelor,z-nr.maxim de monede de un anumit tip
void citire();
int valid(int k);
int solutie();
void afisare(int k);
void BK(int k);
void main()
{citire();
 BK(1);
}
void BK(int k)
{int i;
 for (i=0;i<=z[k];i++)
   {v[k]=i;
    if (valid(k)==1)
      {if (solutie()==1)
          afisare(k);
       else
          BK(k+1);
      }
   }
}
void citire()
{int i;
 ifstream f("c:\\casa\\sanda\\probleme\\cpp\\back\\monede.in");
 f>>S>>n;              //se citeşte suma S şi numărul de monede
 for(i=1;i<=n;i++)
   {f>>w[i];           //se citesc valorile monedelor
    z[i]=S/w[i];}  //z-memorează numărul maxim de monede de un anumit tip, penru  a plati suma S
int valid(int k)
{int i;
 Suma=0;
 for (i=1;i<=k;i++)
   Suma=Suma+v[i]*w[i];
 if ((Suma<=S)&&(k<=n))
   return 1;
return 0;}
int solutie()
{if (Suma==S)
    return 1;
 return 0;}
void afisare(int k)
{int i;
 sol++;cout<<"Solutia :"<<sol<<endl;
 for (i=1;i<=k;i++)
   if (v[i])   cout<<v[i]<<" monede de valoarea "<<w[i]<<endl;
 cout<<endl;}
```

7. Problema rucsacului

Într-un rucsac se poate transporta o anumită greutate maximă G.

O persoană dispune de n obiecte. Pentru fiecare obiect se cunoaşre greutatea şi câştigul pe care persoana îl poate obţine transportând acelst obiect.

Ce obiecte trebuie să transporte persoana respectivă pentru a obţine un câştig maxim.

Datele de intrare se citesc din fişierul RUCSAC.IN astfel:

- linia 1: n G -unde n este numărul de obiecte şi G greutatea maximă admisă de rucsac
- linia i: nume[i] g[i] c[i]

unde: -nume – este numele obiectului
-g – este greutatea obiectului
- c –este caştigul obţinut pentru acel obiect cu i=1,n

Exemplu:

RUCSAC.IN

4 20
pantaloni 5 5
caciula 10 3
camasa 10 7
pantofi 5 2

pentru datele de intrare de mai sus, soluţia optimă este:

Castigul maxim:14
pantaloni 5 5
camasa 10 7
pantofi 5 2

După cum observaţi prolema nu solicită obţinerea tuturor soluţiilor ci determinarea soluţiei optime.

Pentru a determina soluţia optimă vom genera cu metoda backtracking toate soluţiile şi vom reţine dintre acestea doar soluţia cea mai bună. Aceasta presupune să nu afişăm fiecare soluţie ci, în momentul obţinerii unei noi soluţii o vom compara cu soluţia precedentă, în cazul în care câştigul obţinut este mai mare decât precedentul vom reţine această soluţie. Vom considera câştigul iniţial 0.

Vom folosi următoarele variabile:

n-numărul de obiecte
G - greutatea maximă admisă de rucsac
nume[ ][ ] - reţinem renumirea obiectelor
g[ ] - reţinem greutatea fiecărui obiect
c[ ] -reţinem câştigul pentru fiecare obiect
v[ ] -vectorul soluţie:
0-obiectul nu este transportat,
1-obiectul este transportat

s_max -reţine câştigul maxim
sol_max -reţine soluţia maximă

```
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#define MAX 20
int n,v[MAX],sol_max[MAX],g[MAX],c[MAX],s,s_max,G,gr,nr_sol;
char nume[MAX][MAX];
void back(int k);
int valid(int k);
void optim();
void citire();
void afisare();
main()
{citire();
 back(1);
 afisare();                          //afisam solutia optima
}
void back(int k)
{ int i;
 for(i=0;i<=1;i++)
    {v[k]=i;                         //0-obiectul k sete NEselectat, 1-obiectul k este selectat
     if (valid(k))
          if (k==n) optim();         //din multimea solutiilor vom retine doar solutia optima
          else  back(k+1); }
}
int valid(int k)
{ gr=0;
 for(int j=1;j<=k;j++)
   gr=gr+v[j]*g[j];             //-insumam greutatile obiectelor selectate pana la pasul k
 if(gr<=G)  return 1            //verificam daca greutatea cumulata nu depaseste greutatea maxima G
 else    return 0;
}
void optim()
{int s=0;  nr_sol++;
 for(int j=1;j<=n;j++)
   s=s+v[j]*c[j];                          //s-calculam suma câştigurilor obiectelor selectate
 if((nr_sol==0)||(s>s_max))                //daca s>suma maxima, solutia este mai buna
   {s_max=s;                               //retinem solutia in sol_max
    for(j=1;j<=n;j++)
     sol_max[j]=v[j];
   }
}
void citire()
{ ifstream f("RUCSAC.IN");
  f>>n>>G;                          //n-nr. obiecte, G-greutatea maxima
  for (int i=1;i<=n;i++)
     f>>nume[i]>>g[i]>>c[i];        //se citeste greutatea si ponderea fiecarui obiect
  f.close();
}
void afisare()
{ nr_sol++;
 cout<<"Castigul maxim:"<<s_max<<"\n";
 for (int j=1;j<=n;j++)
    if(sol_max[j]) cout<<nume[j]<<" "<<g[j]<<" "<<c[j]<<endl;
 cout<<"\n";
}
```

8.3. Evaluare

TESTUL 1

1. Când este necesar ca în rezolvarea unei probleme să aplicăm metoda backtracking?

2. Ne propunem să generăm toate submulţimile mulţimii {1, 2, 4, 6, 8}. Câte soluţii care obligatoriu conţin elementul 2 şi nu conţin elementul 8 putem genera?
   a.) 8 b.) 6 c.) 16 d.) 7

3. Să se scrie un număr natural n ca sumă de numere neturale nenule distincte.

4. Dacă scriem numărul 9 ca sumă de numere naturale distincte, aplicând metoda backtracking şi obţinem toate soluţiile în ordinea:
   1+2+6, 1+3+5, 1+8, 2+3+4, 2+7, 3+6 şi 4+5, aplicând aceeaşi metodă pentru scrierea lui 12 ca sumă, aplicând exact aceeaşi metodă de generare, Câte soluţii de forma 3+... există?
   a.) 7 b.) 2 c.) 1 d.) 4

TESTUL 2

1. Descrieţi etapele obligatorii de analiză, a unei probleme pe care trebuie să o rezolăm cu metoda backtracking.

2. Presupunând că avem mulţimea {2, 4, 6} şi generăm cu backtracking toate numerele care se pot forma cu aceste cifre în ordine strict crescătoare, nu neapărat în această ordine:
   2, 4, 24, 6, 26, 46, 246. Problema este echivalentă cu a genera:
   a.) permutări de k obiecte
   b.) aranjamente de 10 obiecte luate câte k
   c.) submulţimilor nevide ale unei mulţimi
   d.) partiţiilor unei mulţimi

3. Să se genereze toate numerele care se pot forma cu cifre aflate în ordine strict descrescătoare, din mulţimea {2, 4, 6, 8} .

4. Aveţi n invitaţi la masă. Printre persoanele invitate există câteva care nu se agreează şi nu doriţi să ajungă alături. Determinaţi toate modalităţile de a plasa la masă învitaţii ţinând seama de aceste restricţii. Datele de intrare se ciresc din fişierul back.in astfel:
   linia 1: n -numărul de persoane
   linia 2: p1 p2 -două persoane care nu trebuie sa stea alături
   linia 3: p3 p4 - - " -
   .....
   linia k: pl pm - - " -

8.4. Probleme propuse

1. Să se afişeze toate modalităţile în care poate fi ordonată mulţimea {1,2,...,n} astfel ca numerele 1,2,3 să fie alăturate şi în ordine crescatoare(n>3).

2. Fie dată o mulţime A cu m elemente şi o mulţime B cu n elemente. Să se găsească numarul de permutări al mulţimii AUB, astfel încât primul element al unei astfel de permutări sa fie din A, iar ultimul să fie din B, ştiind că A şi B sunt disjuncte. Să se afişeze toate aceste permutări.

3. O grupă de studenţi trebuie să programeze 4 examene în timp de 8 zile. Afişaţi toate modalităţile în care se poate face aceasta. Dar dacă ultimul examen se va da in mod obligatoriu în ziua a opta?

4. La n clase trebuie repartizaţi m profesori de matematică fiecaruia repartizându-i-se câte m clase (m<=n). Determinaţi toate modalităţile în care se poate face repartizarea.

5. Din 10 persoane, dintre care 6 bărbaţi şi 4 femei se formează o delegaţie alcătuită din 5 persoane dintre care cel puţin doua femei. Afişaţi toate modalităţile în care se poate forma aceasta delegaţie.

6. În câte moduri se poate ordona mulţimea {1,2,..,n} astfel încât fiecare număr divizibil cu 2, şi fiecare număr divizibil cu 3, să aibă rangul divizibil cu 2 şi respectiv 3? Afişaţi toate soluţiile.

7. Pentru întocmirea orarului unei clase de elevi, trebuie să fie programată în fiecare zi, fie o oră de desen din cele două pe săptămână, fie o ora de fizică din cele patru pe săptămână. Afişaţi toate modalităţile de întocmire a orarului.

8. La o petrecere iau parte 7 fete si 8 baieţi. La un anumit dans trebuie să se formeze 4 perechi. În câte moduri se pot forma aceste perechi? Afişaţi toate soluţiile.

9. Un elev are n cărţi de matematică şi altul are m cărţi. În câte moduri pot să schimbe cărţile între ei, o carte în schimbul alteia? Dar dacă schimbă două cărţi în schimbul altora 2? Afişaţi toate soluţiile.

10. Fiind dată o hartă cu n ţări, se cer toate soluţiile de colorare a hărţii, utilizând cel mult 4 culori, astfel încât două ţări cu frontieră comună să fie colorate diferit.

11. Se dau n cuburi numerotate de la 1 la n, de laturi l_i si culori c_i cu care se pot forma turnuri, respectând condiţiile:
    - cuburile din turn au laturile în ordine descrescatoare;
    - cuburi alaturate au culori diferite.

    Folosind k din cele n cuburi, se cere sa se afişeze:
    a. toate turnurile ce se pot forma;
    b. un turn;
    c. un turn de înaltime maximă;

d. toate turnurile de înaltime maximă (fară a genera de 2 ori toate turnurile posibile).