Elemente de combinatorică

(implementări recursive)

I. Generarea permutărilor

```
#include <iostream>
using namespace std;

//n reprezinta numarul de elemente (permutari de n), n<=10
//x este vectorul solutie care va retine permutarea curenta
int n, x[12];

//intoarce true daca s-au completat toate cele n pozitii
bool eSolutie(int k)
{
   //daca am ajuns la pozitia n+1 inseamna ca avem pozitiile de la 1 la n completate corect
   if(k==n+1) return true;
   return false;
}


//verificam daca elementul de pe pozitia k nu a mai fost pus in stanga lui
bool eOk(int k)
{
   for(int i=1; i<k; i++)
   {
       //doua elemente nu pot fi egale intr-o permutare
       if(x[k]==x[i])
          return false;
   }
   return true;
}


//afisam o solutie gasita sub forma unui sir de numere
void print()
{
   for(int i=1; i<=n; i++)
   {
      cout<<x[i]<<" ";
   }
   cout<<endl; //spatiu intre solutii diferite
}
```

```
// functia recursiva de Backtracking care completeaza pozitia k
void BKT (int k)
{
   //daca am gasit o solutie valida (toate cele n elemente sunt plasate)
   if(eSolutie(k))
      print(); //afisam solutia
   else
   {
      //incercam sa punem pe pozitia k fiecare numar de la 1 la n
      for(int i=1; i<=n; i++)
      {
         x[k]=i; //punem provizoriu numarul i pe pozitia k
         if(eOk(k)) //daca numarul nu a mai fost folosit anterior
            BKT(k+1); //trecem la pasul urmator: completam pozitia k+1
      }
   }
}


int main()
{
   //citim numarul de elemente pentru permutari
   cin>>n;

   //pornim algoritmul de Backtracking de la prima pozitie (k=1)
   BKT(1);

   return 0;
}
```

II. Generarea combinărilor

```
#include <iostream>
using namespace std;

//n reprezinta numarul total de elemente din care alegem (n<=10)
//m reprezinta numarul de elemente din fiecare combinare (m<=n)
//x este vectorul solutie
int n, m, x[12];


//intoarce true daca s-au completat toate cele m pozitii
bool eSolutie(int k)
{
   //daca am ajuns la pozitia m+1 inseamna ca avem o combinare completa
   if(k==m+1) return true;
   return false;
}
```

```
// verificam daca elementul curent respecta conditia de ordine crescatoare
bool eOk(int k)
{
   //daca suntem la prima pozitie, orice numar e valid
   if (k==1) return true;
   //elementul curent trebuie sa fie strict mai mare decat cel din stanga lui
   if (x[k]>x[k-1])
      return true;

   return false;
}


//afisam combinarea gasita
void print()
{
   for(int i=1; i<=m; i++)
   {
      cout<<x[i]<<" ";
   }
   cout<<endl; //spatiu intre solutii diferite
}


//functia recursiva de Backtracking care completeaza pozitia k
void BKT (int k)
{
   //daca am gasit o solutie valida (toate cele m elemente sunt plasate)
   if(eSolutie(k))
      print(); //afisam solutia
   else
   {
      //incercam sa punem pe pozitia k numere de la 1 la n
      for(int i=1; i<=n; i++)
      {
         x[k]=i; //punem provizoriu numarul i pe pozitia k
         if(eOk(k)) //daca se pastreaza ordinea crescatoare
            BKT(k+1); //trecem la pasul urmator: completam pozitia k+1
      }
   }
}

int main()
{
   cin>>n>>m;
   // pornim algoritmul de Backtracking de la prima pozitie (k=1)
   BKT(1);
   return 0;
}
```

III. Generarea aranjamentelor

```
#include <iostream>
using namespace std;

//n reprezinta numarul total de elemente din care alegem (n<=10)
//m reprezinta numarul de elemente din fiecare aranjament (m<=n)
//x este vectorul solutie
int n, m, x[12];


//intoarce true daca s-au completat toate cele m pozitii
bool eSolutie(int k)
{
   //daca am ajuns la pozitia m+1 inseamna ca avem un aranjament complet
   if(k==m+1) return true;
   return false;
}


//verificam daca elementul curent nu a mai fost folosit pe pozitiile anterioare
bool eOk(int k)
{
   for(int i=1; i<k; i++)
   {
       //doua elemente nu pot fi egale intr-un aranjament
       if(x[k]==x[i])
          return false;
   }
   return true;
}


//afisam aranjamentul gasit
void print()
{
   for(int i=1; i <=m; i++)
   {
      cout<<x[i]<<" ";
   }
   cout<<endl; //spatiu intre solutii diferite
}
```

```
//functia recursiva de Backtracking care completeaza pozitia k
void BKT (int k)
{
   //daca am gasit o solutie valida (toate cele m elemente sunt plasate)
   if(eSolutie(k))
      print(); //afisam solutia
   else
   {
      //incercam sa punem pe pozitia k numere de la 1 la n
      for(int i=1; i<=n; i++)
      {
         x[k]=i; //punem provizoriu numarul i pe pozitia k
         if(eOk(k)) //daca numarul nu a mai fost folosit anterior
            BKT(k+1); //trecem la pasul urmator: completam pozitia k+1
      }
   }
}

int main()
{
   cin>>n>>m;
   //pornim algoritmul de Backtracking de la prima pozitie (k=1)
   BKT(1);
   return 0;
}
```

IV. Generarea produsului cartezian (pt. m mulțimi, fiecare având elemente de la 1 la n)

```
#include <iostream>
using namespace std;

//n reprezinta valoarea maxima a elementelor (n<=5)
//fiecare multime are elementele 1, 2, ..., n
//m reprezinta numarul de multimi (m<=10)
//x este vectorul solutie
int n, m, x[15];

//intoarce true daca s-au completat toate cele m pozitii
bool eSolutie(int k)
{
   //daca am ajuns la pozitia m+1 inseamna ca avem o solutie completa
   if(k==m + 1) return true;
   return false;
}
```

```
//la produsul cartezian elementele se pot repeta, deci orice alegere e valida
bool eOk(int k)
{
   return true;
}

//afisam vectorul generat din produsul cartezian
void print()
{
   for(int i=1; i<=m; i++)
   {
      cout<<x[i]<<" ";
   }
   cout<<endl; //spatiu intre solutii diferite
}


//functia recursiva de Backtracking care completeaza pozitia k
void BKT (int k)
{
   //daca am gasit o solutie valida (toate cele m elemente sunt plasate)
   if(eSolutie(k))
      print(); //afisam solutia
   else
   {
      //incercam sa punem pe pozitia k numere de la 1 la n
      for(int i=1; i<=n; i++)
      {
         x[k]=i; //punem provizoriu numarul i pe pozitia k
         if(eOk(k)) //fiind mereu true, conditia se indeplineste intotdeauna
            BKT(k+1); //trecem la pasul urmator: completam pozitia k+1
      }
   }
}


int main()
{
   //citim n (numarul de elemente din fiecare multime) si m (numarul de multimi)
   cin>>n>>m;
   //pornim algoritmul de Backtracking de la prima pozitie (k=1)
   BKT(1);
   return 0;
}
```

V. Generarea partițiilor unei mulțimi (de n elemente)

```
#include <iostream>
using namespace std;

//n reprezinta numarul de elemente al multimii {1, 2, ..., n}
//n<=7
//x este vectorul solutie
//x[k] reprezinta indicele submultimii in care se afla elementul k
int n, x[12];

//intoarce true daca s-au distribuit toate cele n elemente
bool eSolutie(int k)
{
   //daca am ajuns la elementul n+1 inseamna ca toate cele n elemente au fost puse intr-o submultime
   if(k==n+1) return true;
   return false;
}

//la partitii, conditia eOk este ca submultimile sa fie create in ordine, fara a sari indici
bool eOk(int k)
{
   //determinam indicele maxim al unei submultimi folosite pentru primele k-1 elemente
   int max_anterior=0;
   for(int i=1; i<k; i++)
   {
      if(x[i]>max_anterior)
         max_anterior=x[i];
   }

   //elementul k poate fi pus intr-o submultime existenta (<= max_anterior)
   //sau poate deschide o submultime noua (max_anterior+1), dar nu mai mare!
   if(x[k]<=max_anterior+1)
      return true;

   return false;
}

//afisam partitia gasita sub o forma usor de citit
void print()
{
   //gasim cate submultimi avem in total in aceasta solutie
   int nr_submultimi=0;
   for(int i=1; i<=n; i++)
   {
      if(x[i]>nr_submultimi)
         nr_submultimi=x[i];
   }
```

```
    //afisam fiecare submultime pe rand
    for(int s=1; s<=nr_submultimi; s++)
    {
        cout<<"{";
        bool primul=true;
        for(int i=1; i<=n; i++)
        {
            if(x[i]==s)
            {
                if(!primul) cout <<",";
                cout<<i;
                primul=false;
            }
        }
        cout<<"} ";
    }
    cout<<endl;
}


//functia recursiva de Backtracking care decide destinatia elementului k
void BKT (int k)
{
    //daca toate cele n elemente au fost repartizate
    if(eSolutie(k))
        print(); //afisam partitia
    else
    {
        //elementul k poate fi pus, teoretic, in submultimi numerotate de la 1 la n
        for(int i=1; i<=n; i++)
        {
            x[k]=i; //punem provizoriu elementul k in submultimea i
            if(eOk(k)) //daca nu am sarit peste numere de submultimi
                BKT(k+1); //trecem la urmatorul element
        }
    }
}


int main()
{
    //citim numarul de elemente
    cin>>n;
    //pornim algoritmul de Backtracking de la primul element (k=1)
    BKT(1);
    return 0;
}
```

VI. Generarea partițiilor unui număr natural nenul n

```
#include <iostream>
using namespace std;

//n reprezinta numarul pe care vrem sa il descompunem (n<=15)
//x este vectorul solutie in care retinem termenii sumei
int n, x[20];

//intoarce true daca suma elementelor pana la pasul anterior a ajuns deja la n
bool eSolutie(int k)
{
   int suma=0;
   for(int i=1; i<k; i++)
      suma+=x[i];
   if (suma==n) return true;
   return false;
}

//verifica daca noul termen adaugat respecta conditiile problemei
bool eOk(int k)
{
   //termenul curent trebuie sa fie mai mare sau egal cu cel din stanga lui
   if(k>1 && x[k]<x[k-1])
      return false;

   //calculam suma tuturor termenilor, inclusiv cel curent
   int suma=0;
   for(int i=1; i<=k; i++)
      suma+=x[i];

   //daca suma a depasit numarul n, pozitia nu este valida
   if(suma>n)
      return false;

   return true;
}

//afisam partitia gasita sub forma de suma
void print(int k)
{
   for(int i=1; i<k; i++)
   {
      cout<<x[i];
      if(i<k-1)
         cout<<"+";
   }
   cout<<endl;
}
```

```
//functia recursiva de Backtracking care completeaza pozitia k
void BKT(int k)
{
   //daca elementele plasate pana la k-1 formeaza deja numarul n
   if(eSolutie(k))
      print(k); //afisam solutia gasita
   else
   {
      //incercam sa punem pe pozitia k valori crescatoare de la 1 la n
      for(int i=1; i<=n; i++)
      {
         x[k]=i; //punem provizoriu valoarea i pe pozitia k
         if(eOk(k)) //daca termenul este valid si nu depaseste suma n
            BKT(k+1); //trecem la pasul urmator: completam pozitia k+1
      }
   }
}

int main()
{
   cin>>n;
   //pornim algoritmul de Backtracking de la prima pozitie (k=1)
   BKT(1);
   return 0;
}
```