II. Recursivitate.

Definiții recursive.

O definiție recursivă foloseşte termenul de definit în corpul definiției. O funcție poate fi recursivă, adică se poate apela pe ea însăşi, în mod direct sau indirect, printr-un lanț de apeluri.

```
/* apel recursiv direct */
void f(int x)
{. . .
  f(10);
 . . .
}
```

```
/* apel recursiv indirect */
void f2(int);    /* declaratie anticipata f2
void f1(int x)                    void f2(int y)
{ . . .                                { . . .
  f2(10);                            f1(20);
  . . .                                   . . .
}                                       }
```

În cazul recursivității indirecte, compilatorul trebuie să verifice corectitudinea apelurilor recursive din fiecare funcție. O declarare anticipată a prototipurilor funcțiilor oferă informații suficiente compilatorului pentru a face verificările cerute.

Exemple.

1^0. numărul natural

- **1 ∈ N**
- dacă **x ∈ N** atunci **succ(x) ∈ N**

2^0. factorialul **n!=**

- **1** dacă **n=0**
- **n*(n-1)!** dacă **n > 0**

```
long fact(int n)
{ return (n? n*fact(n-1) : 1);}
```

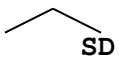
3^0. cel mai mare divizor comun **cmmdc(a, b)=**

- **a** dacă **b=0**
- **cmmdc(b, a%b)** dacă **b > 0**

```
int cmmdc(int a, int b)
{ return (b? cmmdc(b, a%b) : a); }
```

4^0. arborele binar

- arborele vid este arbore binar
- **x** este arbore binar, cu **SS** şi **SD** arbori binari



Orice definiție recursivă are două părți:

- baza sau partea nerecursivă
- partea recursivă

Codul recursiv:

- rezolvă explicit cazul de bază
- apelurile recursive conțin valori mai mici ale argumentelor

Funcțiile definite recursiv sunt simple şi elegante. Corectitudinea lor poate fi uşor verificată.

Definirea unor funcții recursive conduce la ineficiență, motiv pentru care se evită, putând fi înlocuită întotdeauna prin iterație.

Definirea unor funcții recursive este justificată dacă se lucrează cu structuri de date definite tot recursiv.

Recursivitate liniară.

Este forma cea mai simplă de recursivitate şi constă dintr-un singur apel recursiv.

De exemplu pentru calculul factorialului avem:

f(0)=1 cazul de bază

f(n)=n*f(n-1) cazul general

Se vede că dacă timpul necesar execuției cazului de bază este **a** şi a cazului general este **b**, atunci timpul total de calcul este: **a+b*(n-1) = O(n)**, adică avem complexitate liniară.

Recursivitate binară.

Recursivitatea binară presupune existența a două apeluri recursive. Exemplul tipic îl constituie şirul lui Fibonacci:

```
long fibo(int n)
{
  if(n<=2) return 1;
  return fibo(n-1) +fibo(n-2);
}
```

Funcția este foarte ineficientă, având complexitate exponențială, cu multe apeluri recursive repetate.

Se poate înlocui dublul apel recursiv printr-un singur apel recursiv. Pentru aceasta ținem seama că dacă **a**, **b** şi **c** sunt primii 3 termeni din şirul Fibonacci, atunci calculul termenul situat la distanția **n** în raport cu **a**, este situat la distanța **n-1** față de termenul următor **b**.
Necesitatea cunoaşterii termenului următor impune prezența a doi termeni în lista de parametri.

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

long fiblin(int n)
{
  f(0, 1, n);
}
```

Algoritmul rezultat are complexitate liniară.
Complexitatea algoritmului poate îmbunătăți şi mai mult observând că:

$$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n/1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

De unde deducem:

$f_{2n}=f_{n+1}f_n+f_nf_{n-1}=f_n(f_{n+1}+f_{n-1})=f_n(f_n+2f_{n-1})$
$f_{2n-1}=f^2_n+f^2_{n-1}$

Calculul lui f_n se face folosind prima sau a doua dintre relaţiile de mai sus în funcţie de paritatea lui **n** (dacă n este par atunci calculul lui f(n) presupune calculul lui **f(n/2)** şi **f(n/2-1)**, altfel **f((n-1)/2)**şi **f((n+1)/2)**

```
void fiblog(int n, long *f1, long *f2)
{
  long a, b, c, d;
  if(n==1){
    *f1 = 0;
    *f2 = 0;
  }
  else
  {
    fiblog(n/2, &a, &b);
    c = a*a+b*b;
    d = b*(b+2*a);
    if(n%2)
    {
      *f1 = d;
      *f2 = c+d;
    }
    else
    {
      *f1 = c;
      *f2 = d;
    }
  }
}
```

Metoda divizării (Divide et Impera).

Un algoritm recursiv obține soluția problemei prin rezolvarea unor instanțe mai mici ale aceleeaşi probleme.

- se împarte problema în subprobleme de aceeaşi natură, dar de dimensiune mai scăzută
- se rezolvă subproblemele obținându-se soluțiile acestora
- se combină soluțiile subproblemelor obținându-se soluția problemei
- Ultima etapă poate lipsi (în anumite cazuri), soluția problemei rezultând direct din soluțiile subproblemelor.
- Procesul de divizare continuă şi pentru subprobleme, până când se ajunge la o dimensiune suficient de redusă a problemei care să permită rezolvarea printr-o metodă directă.

Metoda divizării se exprimă natural în mod recursiv: rezolvarea problemei constă în rezolvarea unor instanțe ale problemei de dimensiuni mai reduse

```
void DivImp(int dim, problema p, solutie *s){
  int dim1, dim2;
  problema p1, p2;
  solutie s1, s2;
  if(dim > prag){
    Separa(dim,p,&dim1,&p1,&dim2,&p2);
    DivImp(dim1, p1, &s1);
    DivImp(dim2, p2, &s2);
    Combina(s1, s2, s);
  }
  else
    RezDirect(dim, p, s);
}
```

Pentru evaluarea complexității unui algoritm dezvoltat prin metoda *divide et impera* se rezolvă o ecuație recurentă având forma generală: $T(n)=a.T(n/b) + f(n)$, în care $a >= 1$, reprezintă *numărul subproblemelor*, iar n/b este *dimensiunea subproblemelor*, $b > 1$.

$f(n)$ reprezintă costul divizării problemei în subprobleme şi a combinării soluțiilor subproblemelor pentru a forma soluția problemei.

Soluția acestei ecuații recurente este dată de *teorema master*

1^0.dacă $f(n) = O(n^{\log_b a-\varepsilon})$ atunci $T(n) = \Theta(n^{\log_b a})$

2^0.dacă $f(n) = \Theta(n^{\log_b a})$ atunci $T(n) = O(n^{\log_b a} \lg n)$

3^0. dacă $f(n) = \Omega(n^{\log_b a+\varepsilon})$ şi $a f(n / b) \le c f(n),\ c < 1$

atunci $T(n) = \Theta(f(n))$

Exemple

Ridicarea unui număr la o putere întreagă.

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{pentru } x \text{ par} \\ x \cdot x^{n/2} \cdot x^{n/2} & \text{pentru } x \text{ impar} \end{cases}$$

```
double putere(double x, int n) {
  if (n==0) return 1;
  double y = putere(x, n/2);
  if (n%2==0)
    return y*y;
  else
    return x*y*y;
}
```

Se constată că în urma separării se rezolvă o singură subproblemă de dimensiune **n/2.** Aşadar: **T(n)= T(n/2)+1,** care are soluția **T(n)=O(log₂n).**

Căutarea binară

- se compară valoarea căutată **y** cu elementul din mijlocul tabloului **x[m]**
- dacă sunt egale, elementul **y** a fost găsit în poziția **m**
- în caz contrar se continuă căutarea într-una din jumătățile tabloului (în prima jumătate, dacă **y<x[m]** sau în a doua jumătate dacă **y>x[m]**).

Funcția întoarce poziția **m** a valorii **y** în **x** sau **−1**, dacă **y** nu se află în **x**. Complexitatea este **O(log₂n)**

```
 int CB(int i, int j, double y, double x[]) {
  if(i > j) return -1;
  double m = (i+j)/2;
  if(y == x[m]) return m;
  if(y < x[m])
    return CB(i, m-1, y, x);
  else
    return CB(m+1, j, y, x);
}

int CautBin(int n, double x[], double y){
  if(n==0) return -1;
  return CB(0, n-1, y, x);
}
```

Localizarea unei rădăcini prin bisecție

Localizarea unei rădăcini a ecuației **f(x)=0**, separată într-un interval **[a, b]** prin bisecție este un caz particular de căutare binară. Deoarece se lucrează cu valori reale, comparația de egalitate se evită.

```
double Bis(double a, double b, double (*f)(double)){
  double c = (a+b)/2;
  if(fabs(f(c))< EPS || b-a < EPS) return c;
  if(f(a)*f(c) < 0)
    return Bis(a, c, f);
  else
    return Bis(c, b, f);
}
```

Elementul maxim dintr-un vector

Aplicarea metodei divizării se bazează pe observația că maximul este cea mai mare valoare dintre maximele din cele două jumătăți ale tabloului.

```
double max(int i, int j, double *x){
  double m1, m2;
  if(i==j) return x[i];
  if(i==j-1) return ( x[i]>x[j]? x[i]:x[j];);
  int k = (i+j)/2;
  m1 = max(i, k, x);
  m2 = max(k+1, j, x);
  return (m1>m2)? m1: m2;
}
```

Turnurile din Hanoi

Se dau 3 tije:stânga, centru, dreapta; pe cea din stânga sunt plasate **n** discuri cu diametre descrescătoare, formând un turn. Generați mutările care deplasează turnul de pe tija din stânga pe cea din dreapta. Se impun următoarele restricții:

- la un moment dat se poate muta un singur disc
- nu se permite plasarea unui disc de diametru mai mare peste unul cu diametrul mai mic
- una din tije serveşte ca zonă de manevră.

Problema mutării celor **n** discuri din zona sursă în zona destinație poate fi redusă la două subprobleme:

- mutarea a **n-1** discuri din zona sursă în zona de manevră, folosind zona destinație ca zonă de manevră
- mutarea a **n-1** discuri din zona de manevră în zona destinație, folosind zona sursă ca zonă de manevră
- Între cele două subprobleme se mută discul rămas în zona sursă în zona destinație

```
void Hanoi(int n,char s,char m,char d){
  if(n) {
    Hanoi(n-1, s, d, m);
    Printf("%d -> %d\n", s, d);
    Hanoi(n-1, m, s, d);
  }
}
```

Avem satisfăcută ecuația recurentă: **T(n) = 2.T(n-1) + 1,** care se rescrie:

$T(n)=2^2T(n-2)+1)+1=2^2T(n-2)+2+1=2^3T(n-3)+2^2+2+1=$

$2^{n-1}\ T(1)\ +2^{n-2}\ +\ldots+2+1$

$T(n)=2^{n-1}+...+2+1=2^n-1=O(2^n)$

Sortarea prin interclasare (mergesort)

Şirul de sortat se împarte în două subşiruri de dimensiune pe cât posibil egală. Se sortează subşirurile (divizându-le recursiv în alte subşiruri, până când se ajunge la subşiruri de lungime 1, care sunt gata sortate). Subşirurile sortate se recombină într-un singur şir sortat prin interclasare.

```
void merge(int, int, int, double*);
```

```
void msort(int, int, double*);
void mergesort(int n, double *x){ msort(0, n-1, x); }

void msort(int i, int j, double *x){
   if(i<j){
      int k=(i+j)/2;
      msort(i, k-1, x);
      msort(k, j, x);
      merge(i, k, j, x);
   }
}

void merge(int p, int q, int r, double *x){
   int i=p, j=q, k=0;
   double *y = (double*)malloc((r-p+1)*sizeof(double));
   while(i<q && j<=r)
      if(x[i] < x[j])
         y[k++] = x[i++];
      else
         y[k++] =x[j++];
   while(i < q)
      y[k++] = x[i++];
   while(j <= r)
      y[k++] = x[j++] ;
   for(i=p; i<=r; i++)
      x[i] = y[i-p];
   free(y);
}
```

Interclasarea subşirurilor are complexitate liniară **O(n)**. Sortarea prin interclasare se bazează pe ecuația recurentă: **T(n)= 2.T(n/2)+c.n** cu soluția **T(n)=0(n.log$_2$n)**

Partiționare

Partiționarea unui vector în raport cu un pivot, presupune separarea vectorului în două zone (partiții) astfel încât prima partiție să conțină elementele <= cu pivotul, iar cea de-a doua partiție, elementele mai mari ca pivotul.

Considerăm un vector **x** între doi indici **p** şi **r**. Pivotul poate fi oricare element al şirului **x**; noi vom alege ca pivot pe **x[p]**(primul element). Inițial partițiile sunt vide; ele se extind (spre dreapta, respectiv spre stânga), până când se vor atinge. Extinderea unei partiții este împiedicată de prezența unui element care nu aparține partiției. În momentul în care se blochează extinderea ambelor partiții se interschimbă elementele străine partițiilor, după care extinderea partițiilor continuă până la atingere. Funcția întoarce poziția primului element din cea de-a doua partiție.

```
int pivot(double *x, int p, int r){
  double piv = x[p];  // alegere pivot
  int i = p-1;
  int j = r+1;
  double t;
  while(1){
    do { i++;} while (x[i]<=piv);
```

```
    do { j--;} while (x[j]>=piv);
    if(i < j){
       t=x[i];
       x[i]=x[j];
       x[j]=t;
    }
    else
       return j;
  }
}
```

Cel de-al N-lea element.

O metodă evidentă de găsire a celui de-al **N**-lea element ar fi sortarea vectorului şi selectarea elementului din poziția **N**, cu complexitate $O(n \log_2 n)$.
O metodă având o complexitate mai mică partiționează vectorul în raport cu un pivot. Dacă primul element din cea de-a doua partiție este în poziția **j==N**, atunci elementele primei partiții sunt mai mici sau egale cu acesta, deci elementul reprezintă cel de-al **N**-lea element; dacă **j > N** cel de-al N-lea element se află în prima partiție deci va fi căutat recursiv în domeniul **(p, j)**, în caz contrar el se află în cea de-a doua partiție, fiind căutat recursiv în domeniul **(j, r).**

```
double elementN(double* v,int N,int p, int r){
  int j = pivot(v, p, r);
  if(j==N)
    return v[N];
  if(N<j)
    return elementN(v, N, p, j);
  else
    return elementN(v, N, j+1, r);
}
```

Sortarea rapidă (quicksort).

Algoritmul de sortare rapidă, datorat lui Hoare, partiționează vectorul de sortat, în raport cu un *pivot*. Cele două partiții sunt sortate, la rândul lor, prin apeluri recursive.
Astfel şirul **x[p:r]** este partiționat în subşirurile **x[p:q]** şi **x[q+1:r]** astfel încât orice element din prima partiție **x[p:q]** este mai mic sau egal cu un element din cea de-a doua partiție **x[q+1:r]**. În final şirul, în ansamblu, este sortat deoarece partițiile lui sunt sortate.

```
void QS(int, int, double*);
void quicksort(int n, double *x) { QS(0, n-1, x); }

void QS(int p, int r, double *x){
  if(p<r){
    int q = pivot(x, p, r);
    QS(p, q, x);
    QS(q+1, r, x);
  }
}

void qsort(int n, double* v){
  if(n>1)
    QS(0, n-1, v);
```

```
}
```

Complexitatea algoritmului de partiționare este **O(n)**.
Complexitatea algoritmului de sortare depinde de echilibrarea partițiilor. Dacă partițiile sunt dezechilibrate (în cazul cel mai nefavorabil, în care şirul este sortat descrescător cele două partiții sunt de 1, respectiv **n-1** elemente), atunci complexitatea este **$O(n^2)$.**

Înlăturarea recursivității.

În general recursivitatea nu este eficientă. De exemplu generarea recursivă a termenilor şirului Fibonacci are complexitate exponențială, în timp ce soluția iterativă are complexitate liniară.

Algoritmii formulați folosind strategia divide et impera sunt eficienți dacă subproblemele sunt de dimensiuni aproape egale.(pentru subprobleme cu dimensiuni dezechilibrate – de exemplu turnurile din Hanoi, se obține tot complexitate exponențială).

Se preferă soluții recursive în situațiile în care varianta iterativă conduce la algoritmi foarte complicați.

Recursivitatea poate întotdeauna fi transformată în iterație. În acest scop se foloseşte o stivă. Dacă funcția conține un singur apel recursiv şi acesta este ultima instrucțiune (*recursivitate terminală*) transformarea recursivității în iterație presupune următorii paşi:

- se transcrie partea nerecursivă, cu parametrii curenți şi variabilele locale
- se actualizează variabilele şi parametrii corespunzător noii iterații
- se efectuează un salt la începutul părții iterative

Exemplificăm cu afişarea elementelor unei liste înlănțuite:

```
struct nodL {
      int info;
      struct nodL* leg;
};
void afisare(struct nodL* L)
{   if(L){
       printf("%d\n", L->info);
       afisare(L->leg);
    }
}
void afisare(struct nodL* L){
  label 1;
  1:if(L){
      printf("%d\n", L->info);
      L = L->leg;
      goto 1;
    }
}
```

Desființăm saltul necondiționat **goto** folosind un ciclu **while**:

```
void afisare(struct nodL* L) {
```

```
  while(L){
    printf("%d\n", L->info);
    L = L->leg;
  }
}
```

Dacă funcția conține un singur apel recursiv, care nu este terminal, vom folosi o stivă în care se pun parametrii şi variabilele locale. Varianta iterativă a algoritmului recursiv este:

```
while(se fac apeluri recursive)
{
  partea nerecursiva care precede apelul;
  Push(S, parametri);
  Push(S, variabile locale);
  actualizare param.si var.locale pt.urmatorul apel;
}
while(!S_Empty(S)){
  Pop(S);
  parte nerecursiva de dupa apelul recursiv;
}
```

Definim o funcție recursivă care afişează elementele dintr-o listă înlănțuită şi apoi face afişarea elementelor în ordine inversă, începând cu ultimul:

```
void afisare(nodL* L) {
  if(L){
    printf("%d-", L->info);
    afisare(L->leg);
    printf("%d\n", L->info);
  }
}
```

O variantă iterativă este:

```
void afisare(nodL* L) {
  Stiva S;
  while(L){
    printf("%d-", L->info);
    Push(L, S);
    L=L->leg;
  }
  while(!S_Empty(S)){
    L=Pop(S);
    printf("%d\n", L->info);
  }
}
```

Metoda Backtracking.

Metoda backtracking reprezintă una dintre metodele cele mai generale pentru determinarea soluțiilor unei probleme.

Pentru aplicarea metodei este necesar ca soluția problemei să se exprime printr-un n-tuplu $(x_1, x_2, \ldots, x_n)$, în care $x_i \in S_i$, S_i fiind o mulțime finită, izomorfă cu mulțimea $\{1:m\}$. Aceste condiții poartă numele de *restricții* (sau *condiții*) *explicite*, iar tuplii care satisfac restricțiile explicite (produsul cartezian $S_1 \times S_2 \times \ldots \times S_n$) definesc *spațiul soluțiilor problemei*.

De exemplu, pentru generarea permutărilor de n obiecte, mulțimile $S_i=\{1:n\}$ permit formarea a n^n tupli care definesc spațiul soluțiilor.
O soluție vector trebuie să satisfacă o *funcție criteriu* $P(x_1, \ldots, x_n)$, cunoscută şi sub numele de *restricții* (sau *condiții*) *implicite*.
Generarea tuturor tuplilor produsului cartezian $\left(\prod_{i=1}^{n} |S_i| = \prod_{i=1}^{n} m_i = m^n \right)$
şi selectarea celor care satisfac restricțiile implicite, cunoscută sub numele de *metoda forței brute*, nu este aplicabilă, datorită complexității exponențiale şi lipsei oricărei posibilități de optimizare.

Restricțiile implicite acționează ca o *funcție de tăiere* (*limitare*), reducând dimensiunea spațiului soluțiilor problemei, fără a genera toți tuplii.

De exemplu, în generarea permutărilor există restricția implicită ca elementele vectorului soluție să fie distincte, ceea ce reduce numărul posibilităților (soluțiilor) la $n!$.
Restricțiile explicite definesc un arbore (implicit) al spațiului soluțiilor. O soluție reprezintă o cale de la nodul rădăcină la un nod terminal. Căutarea în spațiul stărilor presupune traversarea acestui arbore. În metoda backtracking, această traversare se face *în adâncime*.
Metoda backtracking generează soluția în mod *progresiv*, adăugând la soluția parțială, $x_1, \ldots, x_{k-1}$ o componentă x_k, astfel încât *soluția parțială extinsă* să satisfacă o condiție implicită, cunoscută sub numele de *condiție de continuare*. $P(x_1, \ldots, x_k)$
În generarea stărilor problemei se porneşte din *starea inițială* (*nodul rădăcină*) şi se generează alte noduri. Un nod generat poartă numele de *nod viu*, dacă nu i-au fost generați încă toți descendenții.
Aşadar, am putea defini metoda backtracking ca o generare de noduri în adâncime în arborele spațiului stărilor soluțiilor, folosind funcții de limitare. Complexitatea metodei rămâne exponențială, eficiența ei depinzând în mod semnificativ de eficiența funcțiilor de tăiere.

Datorită complexității exponențiale, metoda backtracking se foloseşte numai în situațiile în care nu dispunem de alte metode

- generarea permutărilor de n obiecte
- generarea tuturor plasărilor a 8 dame pe o tablă de şah, astfel încât să nu se atace
- determinarea tuturor posibilităților de plată a unei sume, folosind anumite tipuri de bancnote
- determinarea tuturor posibilităților de sortare topologică a vârfurilor unui graf

Găsirea unui arbore de acoperire de cost minim într-un graf ponderat se determină eficient prin algoritmii greedy Prim şi Kruskal. Găsirea tuturor arborilor de acoperire poate fi făcută numai prin backtracking.

Dintre toate soluțiile unei *probleme de optimizare*, prezintă interes numai cele care extremizează (maximizează sau minimizează) o *funcție de cost* (sau *funcție obiectiv*) ca de exemplu:

- găsirea tuturor căilor de lungime minimă, de ieşire dintr-un labirint
- colorarea vârfurilor unui graf folosind un număr minim de culori
- găsirea ciclurilor hamiltoniene de cost minim într-un graf ponderat
- debitarea unei bare după nişte repere date, astfel încât restul de tăiere să fie minim
- problema discretă a rucsacului

Presupunem că s-a generat soluția parțială ($x_1, x_2, \ldots, x_{k-1}$) şi dorim s-o extindem prin adăugarea componentei x_k. Apar următoarele situații:

- dacă nu se poate genera x_k, se revine asupra lui x_{k-1} şi se generează o altă valoare
- dacă se poate genera x_k şi
  - sunt îndeplinite condițiile de continuare
  - dacă este soluție – se reține şi se revine căutând alt x_k (altă soluție)
  - dacă nu este soluție – se generează x_{k+1}
- nu sunt îndeplinite condițiile de continuare se generează un nou x_k

int n; - lungimea soluției
int nsol; - numărul de soluții
int x[N]; - vectorul soluție
int inf; – marginea.inferioară a domeniului valorilor componente ale solutiei
int sup; – marginea.superioară a domeniului valorilor componente ale solutiei
int xopt[N][NSOL]; - soluțiile optimale
void BKT(int); - funcție care realizează strategia backtracking în mod recursiv sau iterativ
void RetSol(); - funcție care realizează prelucrările necesare la găsirea unei soluții
int Continuare(int); - funcție care verifică dacă sunt îndeplinite condițiile de continuare
int Solutie(int); - funcție care verifică dacă s-a obținut soluție
int Succesor(int); - funcție care verifică dacă mai există valori disponibile pentru x_k
void Include(int); - conține prelucrările asociate selectării unei componente
void Exclude(int); - conține prelucrările asociate renunțării la o componentă selectată

```
// backtracking recursiv
void BKT(int k){
  if(Solutie(k))
    RetSol();
  else
    for(; x[k]∈S[k] && P(x[0],…,x[k]);)
      BKT(k+1);
}
void BKT(int k){
  if(Solutie(k))
    RetSol();
  else
    while(x[k]=Successor(k))
      if(Continuare(k))
        BKT(k+1);
}

void BKT (int k) {
  if (Solutie(k))
```

```
        RetSol();
  else
     { x[k] = inf-1;
       while(Succesor(k))
          if(Continuare(k))
             BKT(k+1);
     }
}
void BKT(int k) {
  int i;
  for (i=inf;i<=sup;i++) {
     x[k] =i;
     if (Continuare(k))
       if (k==n)
          RetSol();
       else
          BKT(k+1);
  }
}
void BKT(int k) {
  int i;
  if ( k > n)
      RetSol();
  else
   for (i=inf;i<=sup;i++) {
       x[k] =i;
       if (Continuare(k)) {
           Include (k);
           BKT(k+1);
           Exclude (k);
       }
   }
}
//backtracking iterativ
void BKT(){
  for(int i=1;i<=n;i++)  x[i]=inf-1;
  int k=1;
  while(k>0)
    if(k>n){           //avem solutie
      RetSol();
      k--;             //cauta alta solutie
    }
    else               //solutie incompleta
      if(x[k]<sup){    //mai sunt valori pt.x[k]
        x[k]++;        //urmatoarea valoare
        if(Continuare(k)) //acceptabila
          k++;         //se trece la aflare x[k+1]
      }
      else           //s-au epuizat posibilitatile
```

```
        { x[k]=inf-1;
          k--;          //se revine la x[k-1]
        }
}

void BKT () {
  int k=1, exista, corect;
  x[k] = inf-1;
  while(k > 0) {
    do { exista = Succesor(k);
         corect = Continuare(k);
    } while(exista && !corect);
    if(exista)
      if(Solutie(k))
         RetSol();
      else
         x[++k] = inf-1;
    else
      k--;
  }
}
}

void BKT(){
  int k=1;
  while(k){
     if(Solutie(k)){
       RetSol();
       nsol++;
       k--; //pas inapoi pt gen. alta solutie
     }
     else
       if(x[k]<sup){
          x[k]++;
          if(Continuare(k))
             k++;
       }
       else
           { x[k]=inf-1;//nu se poate avansa
             k--;
           }
  }
}
```

Exemple de probleme care se rezolvă folosind backtracking.

1. Pentru **n** dat, să se genereze permutările de **n** obiecte.

```
#include <stdio.h>
#include “backtrack.h”
void main(){
```

```
  scanf("%d", &n);
  BKT(0);
}
int Solutie(int k){
  return k> n-1;
}
void RetSol(){
  for(int i=0; i<n; i++)
    printf("%d ", x[i]);
  printf("\n");;
}
int Continuare(int k){
  for(int i=0; i< n; i++)
    if(x[k]== x[i]) return 0;
  return 1;
}
void BKT(int k){
  if(Solutie(k))
    RetSol();
  else
    for(int i=1; i<= n; i++){
      x[k]=i;
      if(Continuare(k))
        BKT(k+1);
    }
}
```

2. Să se genereze toate posibilitățile de plasare a 8 regine pe tabla de şah, astfel ca acestea să nu se poată ataca între ele. (Reginele se pot ataca pe linii, pe coloane sau pe diagonale).

Întrucât fiecare regină este plasată pe o coloană diferită **k=0:7**, reginele nu se vor ataca pe coloane. Poziția unei regine pe tablă va fi dată de perechea (**k, x[k]**), deci soluția poate fi reprezentată printr-un vector **x** de lungime **8.** Condiția de neatacare pe linii impune ca **x[k]≠ x[i]** pentru **i=0:k-1**, iar pentru diagonale, condițiile ca triunghiurile dreptunghice având ca vârfuri ascuțite (**k, x[k]**) şi (**i,x[i]**) să nu fie isoscele, adică: **k-i≠ x[k]-x[i]**, respectiv **k-i≠ x[i]-x[k].** Cele două condiții pot fi comasate întruna singură **k-i≠ |x[k]-x[i]|**.

```
int Continuare(int k){
  for(int i=0; i<n; i++)
    if(x[k]==x[i]||k-i==abs(x[k]-x[i]))
      return 0;
  return 1;
}
void RetSol(){
  for(int i=0; i<8; i++){
    for(int j=0; j<8; j++)
      if(j==x[i])
        printf("x");
```

```
      else
        printf("0");
    printf("\n");
  }
}
```

3. Să se genereze o soluție de acoperire a unei table de şah de dimensiune **nxn** de către un cal, care porneşte de pe tablă dintr-o poziție inițială **(x0,y0)** şi urmează o deplasare specifică în L, fără a ieşi în afara tablei de şah şi fără a trece prin poziții pe care le-a parcurs deja.

Soluția este de lungime fixă n^2 şi este reprezentată printr-o matrice, completată cu numerele de ordine ale mutărilor, începând cu poziția inițială, numerotată cu 1. O poziție liberă pe tablă este marcată cu 0. Avansarea pe tablă dintr-o poziție curentă (**xc,yc**) într-o poziție următoare (**xu,yu**) indică 8 posibilități virtuale, dacă mutarea este pe tablă: **0<=xu,yu<=n-1**, şi dacă nu s-a trecut deja prin acea poziție **tabla[xu][yu]==0.** Coordonatele celor 8 posibile deplasări, relative la poziția curentă sunt păstrate în tablourile **dx** şi **dy**.

```
int x[N];
int n;
int nsol=0;
int tabla[N][N];
```

Funcția **Continuare(x,y)** validează mutarea cu coordonatele **(x,y).**

```
int Continuare(int x, int y){
    return nsol==0 && x>=0 && x<n &&
                      y>=0 && y<n &&
                      tabla[x][y]==0;
}
```

Funcția **Solutie(k)** testează dacă s-au efectuat n^2 deplasări pe tabla de şah.

```
bool Solutie(int k){
  return k > n*n;
}
```

Funcția **RetSol()** afişează matricea **tabla[][].**
Ocuparea unei poziții are ca efect secundar marcarea numărului mutării pe tablă.

```
void Ocupa(int k, int x, int y){tabla[x][y]=k;}
```

Revenirea dintr-o poziție selectată anterior, determină eliberarea acesteia de pe tablă:

```
void Elibereaza(int x, int y){tabla[x][y]=0;}
void BKT(){
  int xu,yu;
  if(Solutie(k)){
     nsol++;
     RetSol();
  }
  else
     for(int i=0; i<8; i++){
        xu=xc+dx[i];
        yu=yx+dy[i];
        if(Continuare(xu,yu)){
           Ocupa(k,xu,yu);
```

```
            BKT(k+1,xu,yu);
            Elibereaza(xu,yu);;
         }
      }
}
```

Funcția **main()** citeşte n şi poziția inițială **(x0,y0)**, inițializează tabla ca liberă, plasează calul în prima poziție şi apelează funcția **BKT()** pentru a genera celelalte mutări.

```
#include <stdio.h>
#define N 8
#include "backtrack.h"
int dx[8]={2,1,-1,-2,-2,-1,1,2};
int dy[8]={1,2,2,1,-1,-2,-2,-1};
int main(){
  int x0,y0;
  scanf("%d", &n);
  scanf("", &x0, &y0);
  //initializare tabla
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      tabla[i][j]=0;
  tabla[x0][y0]=1;
  BKT(2,x0,y0);
}
```

Backtracking optimizat.

Eficiența metodei backtracking depinde de mai mulți factori:

- timpul necesar generării următoarei componente **x[k]**
- numărul candidaților care satisfac restricțiile explicite (dimensiunea spațiului soluțiilor)
- timpul necesar verificării funcției criteriu (restricțiile implicite)
- numărul candidaților care satisfac restricțiile implicite

Funcția criteriu (numită şi funcție de limitare) este eficientă, dacă reduce în mod substanțial numărul de noduri generate. Este posibil ca o funcție de limitare eficientă să necesite un timp mare de evaluare, aşa că vom prefera în general reducerea timpului de calcul.
Pentru multe probleme se cunosc metode de limitare simple şi eficiente. Selectarea următoarei componente ar permite considerarea componentelor în orice ordine, dar o simplă *rearanjare a componentelor* ar putea reduce numărul selecțiilor posibile.
Mulțimea **A** conține **n** întregi pozitivi $a_0, a_1, \ldots, a_{n-1}$. Găsiți submulțimile lui **A**, a căror sumă este egală cu o valoare dată **S**.
Vom alege vectorul soluție de lungime fixă **n**, cu următoarea semnificație: dacă **x[k]=0**, atunci componenta **a[k]** nu a fost selectată în soluție.Condiția de continuare în pasul **k** impune:

$$\sum_{j=1}^{k} a_j x_j + \sum_{j=k+1}^{n-1} a_j \geq S$$

(în caz contrar, chiar dacă am selecta toate componentele rămase, nu am reuşi să obținem valoarea **S**).
Dacă în prealabil, sortăm crescător componentele, putem îmbunătăți condiția de continuare prin

$$\sum_{j=1}^{k} a_j x_j + a_{k+1} \leq S$$

(în caz contrar nu am mai avea soluție cu selecția făcută până acum). Vom evita calculul în fiecare pas al sumelor $\sum_{j=0}^{k} a_j x_j$ şi $\sum_{j=k+1}^{n-1} a_j$ punând aceste valori în lista de parametri ai funcției **BKT()**. Aceasta implică $a_0 < s$ şi $\sum_{j=1} a_j \geq s$.

Apelul inițial $BKT\left(0, 0, \sum_{j=0}^{n-1} a_j\right)$ presupune calculul în prealabil al ultimului parametru.

În funcția **BKT()** vom considera separat cele două situații: **x[k]=1** selectăm componenta **a[k]** şi **x[k]=1** nu selectăm componenta. Condițiile de continuare, ca şi stabilirea dacă s-a găsit soluție se formulează explicit, fără a mai defini funcții suplimentare.

Probleme propuse.

Divide et impera

1. Pentru evaluarea polinomului $p_n(x)=a_0+a_1x+\ldots+a_nx^n$ prin metoda divide et impera, se grupează termenii polinomului sub forma:
a) pentru **n** – par: **n=2k**

$$p_n(x)=a_0+a_2x^2+\ldots+a_{2k}x^{2k}+x(a_1+a_3x^2+\ldots+a_{2k-1}x^{2k-2})= b_0+b_1y+\ldots$$
$$+b_ky^k+\sqrt{y}(c_0+c_1y+\ldots+c_{k-1}y^{k-1})$$

b) pentru **n** – impar: **n=2k+1**

$$p_n(x)=a_0+a_2x^2+\ldots+a_{2k}x^{2k}+x(a_1+a_3x^2+\ldots+a_{2k+1}x^{2k})= b_0+b_1y+\ldots$$
$$+b_ky^k+\sqrt{y}(c_0+c_1y+\ldots+c_ky^k)$$

Se evaluează direct numai polinoamele de grad 0 şi 1. Scrieți o funcție cu semnătura:

```
double poly(int n, double *a, double x);
```

care evaluează polinomul prin metoda descrisă.
Exemplu: $1-x+2x^2-4x^3+3x^4+x^5=1+2x^2+3x^4-x(1+4x^2-x^4)=$ **2p**
$$= 1+2y+3y^2-\sqrt{y}(1+4y-y^2)= 1+3y^2+y(2)-\sqrt{y}[1-y^2+y(4)] =$$
$$= 1+3z+\sqrt{z}(2) \qquad 1-z+\sqrt{z}(4)$$

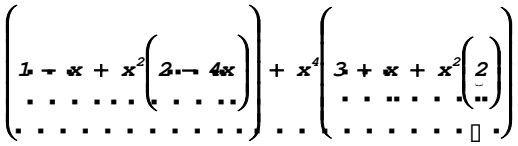
2. Pentru evaluarea polinomului $p_n(x)=a_0+a_1x+\ldots+a_nx^n$ prin metoda divide et impera, se grupează termenii polinomului sub forma:
a) pentru **n** – par: **n=2k**

$$p_n(x)=a_0+a_1x+\ldots+a_kx^k+x^{k+1}(a_{k+1}+a_{k+2}x^2+\ldots+a_{2k}x^{k-1})= a_0+a_1x+\ldots$$
$$+a_kx^k+x^{k+1}(b_0+b_1x+\ldots+b_{k-1}x^{k-1})$$

b) pentru **n** – impar: **n=2k+1**

$$p_n(x)=a_0+a_1x+\ldots+a_kx^k+x^{k+1}(a_{k+1}+a_{k+2}x+\ldots+a_{2k+1}x^k)= a_0+a_1x+\ldots$$
$$+a_kx^k+x^{k+1}(b_0+b_1x+\ldots+b_kx^k)$$

Se evaluează direct numai polinoamele de grad 0 şi 1.
Scrieți o funcție cu semnătura: `double poly(int n, double *a, double x);`
care evaluează polinomul prin metoda descrisă.
Exemplu: $p(x)=1-x+2x^2+-4x^3+3x^4+x^5+2x^6 = 1-x+2x^2-4x^3)+x^4(3+x+2x^2) =$



Se dă un vector **x** cu **n** elemente şi un întreg **0≤ k<n**. Să se determine, utilizând o strategie divide et impera, cel de-al **k**-lea cel mai mic element fără a sorta vectorul **x.**

3. Scrieti un algoritm de tip divide et impera care stabileste un orar de disputare a unui turneu de tenis cu **n** concurenți, în care fiecare doi concurenți se întâlnesc o singură dată. Se vor considera două situații:
- **n** este o putere a lui 2. (Numărul de zile în care se desfățoară turneul este **n-1**).
- **n** este oarecare. (În acest caz numărul de zile este **n-1** dacă **n** este par sau **n** dacă **n** este impar).

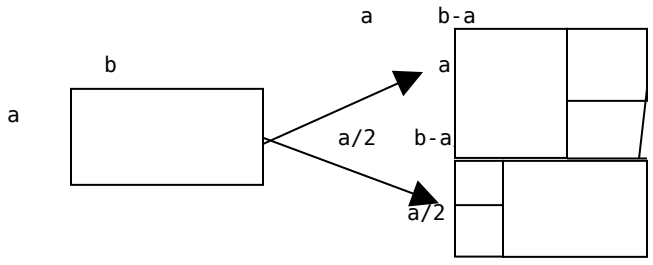
4. Să se modifice algoritmul de căutare binară, astfel ca în cazul în care valoarea căutată nu se află în tabloul **x**, funcția să întoarcă poziția în care trebuie înserată această valoare, adică indicele **j** pentru care: **x[j] <= y < x[j+1]**.

5. La un turneu de tenis, la care participă **n** jucători, fiecare jucător joacă cu ceilalți **n−1**. Bazându-ne pe rezultatele directe ale jocurilor între ei, să se stabilească un clasament folosind strategia divide et impera.

6. Un tablou **A** conține **n** elemente **A[1] < A[2] <...< A[n]**. Scrieți un algoritm cu eficiență mai bună ca **O(n)**, care determină, dacă există, un element **A[i] = i**. Care este complexitatea algoritmului în cazul cel mai nefavorabil.

7. Se dă un vector **x** cu **n** elemente şi un întreg **1 ≤ k ≤ n**. Să se determine, utilizând o strategie divide et impera, cel de-al **k**-lea cel mai mic element fără a sorta vectorul **x**.

8. Un dreptunghi de dimensiuni a şi b, având laturile întregi în relația **a−b < a < b**, poate fi descompus în două pătrate şi un dreptunghi în două moduri posibile:



Dreptunghiul rezultat este descompus în acelaşi mod până se ajunge la un dreptunghi care nu mai satisface relația dintre laturi.
Să se determine şirul de descompuneri în urma căruia rezultă un număr minim de figuri care nu mai pot fi descompuse.

Descompunerea minimă va fi reprezentată printr-un şir de pătrate şi un dreptunghi.
De exemplu: a=21,b=34 se obţin 5 pătrate 21,13,4,8,1 şi un dreptunghi 1 7
Indicaţie: Se defineşte o funcţie recursivă de descompunere (cu parametri laturile unui dreptunghi, numărul de figuri nedecompozabile şi dimensiunile acestora), care determină (dacă dreptunghiul poate fi descompus) cele două modalităţi de descompunere şi se selectează descompunerea cu număr minim de componente.

9. O bucată de tablă de dimensiuni a x b are practicate n găuri cu centrele având coordonatele (x[i],y[i]),raportate la colţul stânga jos, considerat ca origine. Diametrele găurilor se consideră neglijabile.
Determinaţi, prin coordonatele a două colţuri opuse, folosind o strategie divide et impera, bucata dreptunghiulară de arie maximă, care nu conţine găuri, ce se poate decupa prin tăiere pe direcţii paralele cu laturile.

10. Într-un parc de formă dreptunghiulară cu dimensiunile **a x b** se află **n** copaci, având coordonatele (x_i, y_i), **i=0:n-1**, în raport cu originea – colţul stânga jos al parcului.
Găsiţi toate amplasamentele posibile pentru un teren de tenis, având dimensiunile **u x v**, fără a tăia în acest scop vreun copac. Un amplasament este un dreptunghi care nu include copaci în interiorul său, ci numai pe margini, caracterizat prin dimensiunile sale şi coordonatele colţului stânga jos, raportate la acelaşi sistem de coordonate.

Indicaţie: Într-un teren dreptunghiular, dat prin dimensiuni şi coordonatele colţului stânga-jos se verifică dacă există copaci. Prezenţa unui copac împarte mai întâi terenul printr-o orizontală în două terenuri (şi apoi printr-o verticală în alte două terenuri), în care, în mod recursiv se caută copaci. În momentul în care se obţine un teren fără copaci, se verifică dacă dimensiunile lui pot forma un teren de tenis, şi în caz afirmativ se memorează dimensiunile lui şi coordonatele colţului stânga-jos.

11. Pentru calculul celui mai mare divizor comun a două numere se foloseşte algoritmul lui Euclid. Pentru a calcula cmmdc a **n** numere plasate într-un tablou, folosind metoda divide et impera, se împarte tabloul în două jumătăţi şi se calculează cmmdc al celor doi divizori comuni ai celor două jumătăţi ale tabloului. Calculaţi cmmdc al elementelor tabloului şi evaluaţi complexitatea.

Backtracking

1. Pe o tablă de şah de dimensiune n x n se dau coordonatele (x,y) ale unui rege, ale unui cal şi a p obstacole.
Scrieţi un program care să-l ajute pe cal să ajungă la rege şi să se întoarcă înapoi în pozitia de plecare, generându-i toate traseele posibile, urmând mersul specific în L al calului, evitând obstacolele, fară a ieşi de pe tablă şi fară a trece prin pozitii pe care le-a parcurs deja. Programul va genera fiecare traseu şi lungimea sa

2. După o cercetare atentă a inscriptiilor obscene de pe ziduri de tipul IONUT + MARIA = COSTEL, Vasilică a ajuns la concluzia că acestea nu reprezintă triunghiuri conjugale, ci simple operaţii de adunare în baza 10 codificate. Scrieţi un program carte să-l ajute pe Vasilică să descifreze misterioasele adunări, stabilind corespondenţele între litere şi cifre.

3. Acoperirea cu noduri a arcelor unui graf.Fiind dat un graf neorientat `G=(V,M)`, să se determine numarul minim de vârfuri (o submulțime a mulțimii `V`) care acoperă toate muchiile grafului (mulțimea `M`).

Indicatie: Vectorul soluție contine numere de vârfuri şi are o lungime variabila, mai mică sau egală cu n (n fiind numarul de vârfuri din graf).
Conditia de acceptare a unui vârf în pozitia k din vectorul soluție: să nu fi fost selectat anterior (diferit de `x[0],..x[k-1]`)
S-a ajuns la o soluție atunci când au fost acoperite toate muchiile din graf.
Se va stabili mai întâi cum se tine evidenta muchiilor acoperite şi respectiv neacoperite încă din graf.

Exemplu de date: Graful cu 7 noduri şi cu muchiile : (1,2), (2,3), (3,4), (3,5), (4,5), (4,6), (5,6), (4,7). Soluția optimă foloseste 3 noduri: x[1]=2, x[2]=4, x[3]=5.

4. Pentru ca echipa națională de fotbal să obțină rezultate mai bune, Iordănescu a recurs la serviciile unui informatician, care a propus trecerea celor 11 jucatori pe alte posturi decât cele pe care evoluează în mod obişnuit, în scopul imbunătățirii randamentului global al echipei.
Pentru aceasta s-a făcut o evaluare a valorii fiecărui jucător printr-un procentaj şi pentru fiecare jucător s-a evaluat (tot procentual), randamentul utilizării jucătorului pe fiecare post.

Folosind aceste date, să se stabilească o alocare a jucă torilor pe posturi astfel încât randamentul global al echipei să fie maxim.

5. Pe o casetă trebuiesc înregistrate `n` melodii. Pentru fiecare melodie se cunoaşte durata în secunde. Determinații toate posibilitățile de plasare ale melodiilor pe cele două fețe, astfel ca diferența duratelor celor două fețe să fie minimă.

6. Generați toate partițiile unui număr întreg **`n`**.
De exemplu: **`5=5=4+1=3+2=3+1+1=2+2+1=2+1+1+1=1+1+1+1+1`**

7. O matrice `nxn` are elemente întregi 1, 2 şi 3. În matrice se introduc n elemente 0. Determinați toate posibilitățile de plasare ale acestora, ştiind că pot fi înlocuite numai elemente 1 şi 2, şi că nu pot apare mai multe elemente zero pe aceeaşi linie, coloană sau diagonală.

8. Un alfabet conține `v` vocale şi `c` consoane. Să se genereze toate cuvintele de lungime `n`, care nu conțin 3 vocale sau 3 consoane alăturate. Datele citite sunt: `n`-lungimea cuvintelor, `v`-numărul de vocale, `voc`-tabelul vocalelor, `c`-numărul consoanelor şi `cons`-tabelul consoanelor.

9. Se consideră o bară de lungime **`M`** şi **`n`** repere avand lungimile: $a_0 < a_1 < ... < a_{n-1}$
Să se genereze toate modurile distincte în care poate fi tăiată bara, fără a rămâne rest de tăiere, putând folosi un reper o singură dată.

10. Se consideră o bară de lungime **`M`** şi **`n`** repere avand lungimile: $a_0 < a_1 < ... < a_{n-1}$
Să se genereze toate modurile distincte în care poate fi tăiată bara, fără a rămâne rest de tăiere, putând folosi un reper de mai multe ori.

11. Se consideră o bară de lungime M şi n repere avand lungimile: $a_0 < a_1 < ... < a_{n-1}$

Să se genereze toate modurile distincte în care poate fi tăiată bara, putând folosi un reper de mai multe ori, astfel încât restul rezultat din tăiere sa fie minim.

12. Acoperirea cu arce a nodurilor unui graf Un graf cu costuri poate fi acoperit cu arce prin mai multi arbori (grafuri fara cicluri). Un arbore acopera un graf daca atinge toate nodurile sale cu arce din graful initial.

Indicație: Pentru a determina arborele de cost minim metoda backtracking generează toți arborii de acoperire posibili, calculează costurile lor (ca suma a costurilor arcelor), le compară şi reține arborele cu cost minim.
O soluție a problemei este un vector de arce (de perechi de noduri) şi poate fi redus la un vector de predecesori. In acest vector de întregi **x, x[k]** reprezintă predecesorul nodului **k**, deci arcele din arborele căutat sunt de forma **(x[i],i)**. Vectorul **x** are exact **n-1** omponente.
Condițiile de acceptare a unei valori (între 1 si n) pentru **x[k]** sunt:
- **x[k]** diferit de **k**
- să existe arc între **x[k]** si **k**
- arcul **(x[k] - k)** sa nu fie deja inclus în arborele de cost minim

Exemplu: Graful neorientat cu 4 noduri si cu arcele **(1,2)=1,(1,4)=2,(2,3)=2**, **(2,4)=1, (3,4)=1**.
Pentru acest arbore exista 8 arbori de acoperire diferiti, dintre care 3 au costul 5, 4 au costul 4 si unul are costul 3. Arborele de cost minim este **1-2,2-4,3-4** cu 3 arce de cost 1 fiecare. Vectorul solutie va fi **x[1]= 2, x[2]=4, x[3]=4**

13. Acoperirea cu noduri a arcelor unui graf.Fiind dat un graf neorientat **G=(V,M)**, sa se determine numarul minim de vârfuri (o submultime a multimii **V**) care acopera toate muchiile grafului (multimea **M**).
Indicatie: Vectorul solutie contine numere de vârfuri si are o lungime variabila, mai mica sau egala cu n (n fiind numarul de vârfuri din graf).
Conditia de acceptare a unui vârf în pozitia k din vectorul solutie: sa nu fi fost selectat anterior (diferit de **x[0],..x[k-1]**)
S-a ajuns la o solutie atunci când au fost acoperite toate muchiile din graf.
Se va stabili mai întâi cum se tine evidenta muchiilor acoperite si respectiv neacoperite înca din graf.
Exemplu de date: Graful cu 7 noduri si cu muchiile :
(1,2), (2,3), (3,4), (3,5), (4,5), (4,6), (5,6), (4,7)
Solutia optima foloseste 3 noduri: **x[1]=2, x[2]=4, x[3]=5**.

14. Pentru repartizarea celor **n** studenti (**n** par) dintr-o grupă în subgrupe formate de câte 2 studenți se foloseşte un tablou **P[n][n]** în care **P[i][j]** este preferința studentului **i** pentru a lucra cu studentul **j**. Ponderea perechii **(i, j)** se exprimă prin **P[i][j]*P[j][i]**. Se cere sa împerechem studenții astfel încât suma ponderilor perechilor să fie maximizată.