

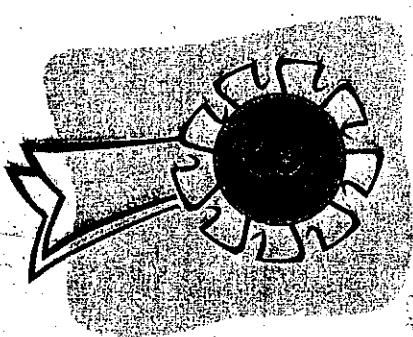
*Dana Lica*

*Mircea Pasoi*

# INFORMATICA

## FUNDAMENTELE PROGRAMARII

Culegere de probleme – Pascal și C/C++  
pentru clasa a XI-a



Toate drepturile asupra acestei lucrări aparțin editurii L&S SOFT.

Reproducerea integrală sau parțială a textului din această carte este posibilă doar cu acordul scris al editurii L&S SOFT.

**Editura L&S SOFT:**

Adresa: Str. Stănjeneilor nr. 8, bl. 29, sc. A, et. 1, apt. 12, sector 4, București.

Telefon: 021-3321315; 021-6366344; 0722-530390; 0722-573701;

Fax: 021-3321315;

E-mail: tsorin@ls-infomat.ro;

Web Site: www.ls-infomat.ro.

**Descrierea CIP a Bibliotecii Naționale a României**

**LICA, DANA**

**Fundamentele programării : culegere de probleme de informatică - Pascal și C++ pentru clasa a XI-a / Dana Lica, Mircea Pașoi. - București : Editura L& S Soft, 2006**

ISBN (10) 973-88037-2-1 ; ISBN (13) 978-973-88037-2-5

Mircea Pașoi

004.42(075.35)(076)

Tiparul executat la **S.C. Lumina TIPO s.r.l.**

Str. Luigi Galvani nr. 20 bis, Sector. 4, București, tel./fax: 211.32.60; tel: 212.29.27

E-mail: office@luminatipto.com; www.luminatipto.com

## CUPRINS

### Capitolul 1

#### Structuri de date

1.1 Structuri de date alocate dinamic – Liste liniare	5
1.1.1 Teste cu alegere multiplă și duală	5
1.1.2 Probleme rezolvate	19
1.1.3 Probleme propuse	31
1.2 Arbo里 și arborescente	37
1.2.1 Concepte teoretice fundamentale	37
1.2.2 Teste cu alegere multiplă și duală	43
1.2.3 Probleme rezolvate	46
1.2.4 Probleme propuse	55
1.3 Structuri de date avansate	58
1.3.1 Tabele de dispersie-hash	58
1.3.2 Arbo里 de intervale	61
1.3.3 Arbo里 indexați binar	68
1.3.4 Arbo里 eficienți de căutare-treapăuri	71
1.3.5 Probleme propuse	74

### Capitolul 2

#### Teoria grafurilor

2.1 Noțiuni introductive	79
2.1.1 Terminologie	79
2.1.2 Moduri de reprezentare la nivelul memoriei	82
2.2 Grafuri orientate și neorientate	87
2.2.1 Teste cu alegere multiplă și duală	87
2.2.2 Probleme rezolvate	91
2.2.3 Probleme propuse	124
2.3 Probleme și algoritmi avansați pe grafuri	134
2.3.1 Probleme rezolvate	134
2.3.2 Probleme propuse	160

## Capitolul 3

### Metode de programare

3.1 Metoda backtracking.....	169
3.1.1 Probleme rezolvate.....	169
3.1.2 Probleme propuse.....	186
3.2 Metoda Divide et Impera.....	191
3.2.1 Probleme rezolvate.....	191
3.2.2 Probleme propuse.....	201
3.3 Metoda programării dinamice.....	204
3.3.1 Probleme rezolvate.....	204
3.3.2 Probleme propuse.....	217
3.4 Metoda Greedy.....	221
3.4.1 Probleme rezolvate.....	221
3.4.2 Probleme propuse.....	232
3.5 Probleme de concurs.....	234
3.5.1 Probleme rezolvate.....	234
3.5.2 Probleme propuse.....	255
<b>Indicatii si raspunsuri.....</b>	<b>265</b>

## CAPITOLUL 1

### Structuri de date

#### 1. 1. Structuri de date alocate dinamic - Liste liniare

##### 1.1.1 Teste cu alegere multiplă și duală

1. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înălțuite:

```
type point^=^nod;
nod=record
    inf : integer;
    leg : point; end;
```

```
struct point
{ int inf;
  point *leg;
};
```

Identificați care dintre următoarele funcții returnează un număr întreg ce reprezintă prima valoare strict pozitivă dintr-o listă simplu înălțuită. Adresa de început a acesteia este trimisă la apel prin intermediul parametrului *p*. În cazul în care lista nu conține decât valori negative, funcția returnează valoarea 0.

- a)
- function V(p:point):integer;
begin
V:=0;
while (p<>nil) do begin
 if (p^.inf>0) then
 begin V:=p^.inf; exit; end;
 p:=p^.leg;
end;
end;
- b)
- function V(p:point):integer;
begin
V:=0;
while (p^.inf<0) do p:=p^.leg;
V:=p^.inf;
end;
- c)
- function V(p:point):integer;
begin
while ((p<>nil)and(p^.inf<=0)) do
 p:=p^.leg;
if (p)
 return p->inf;
return 0;
end;

```

if (p==nil) then V:=p^.inf
else V:=0;
end;

(d)
function V(p:point):integer;
var x:integer;
begin
x:=0;
while (x=0) do begin
  if (p^.inf>0) then x:=p^.inf;
  p:=p^.leg;
end;
V:=x;
end;

```

2. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point^=nod;
  nod=record
    inf : integer;
    leg : point; end;

```

Identifică care dintre următoarele funcții calculează numărul valorilor nule dintr-o listă simplu înlăntuită. Adresa de început să acesteia este trimisă la apel prin intermediul parametrului *p*.

a)

```

function Nr(p:point):integer;
var x:integer;
begin
x:=0;
while (p<>nil) do begin
  if (p^.inf=0) then inc(x);
  else p:=p^.leg;
end;
Nr :=x;
end;

```

b)

```

function Nr(p:point):integer;
begin
if p=nil then Nr:=0
else
  if (p^.inf=0) then
    Nr:=Nr(p^.leg)+1
  else Nr:=Nr(p^.leg)
end;

```

c)

```

function Nr(p:point):integer;
begin
if p=nil then Nr:=0
else
  if (p^.inf=0) then
    Nr:=Nr(p^.leg)+1
  else Nr:=Nr(p^.leg)
end;

```

d)

```

int V(point *p)
{
  int x=0;
  for (;!x; p=p->leg)
    if ((p->inf>0)
        x=p->inf;
  return x;
}

```

```

else
  if ((p->inf=0) then
    Nr:=Nr(p->leg)+1
end;

(d)
function Nr(p:point):integer;
var x:integer;
begin
x:=0;
repeat
  if ((p->inf=0) then inc(x);
  p:=p->leg;
until p=nil;
Nr :=x;
end;

```

(d)

```

int Nr(point *p)
{
  int x=0;
  do {
    x+=!p->inf;
    p=p->leg;
  }while (!p!=NULL);
  return x;
}

```

3. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point^=nod;
  nod=record
    inf : integer;
    leg : point; end;

```

```

struct point
  { int inf;
    point *leg;
  };

```

Funcția *Vm* primește, prin intermediul parametrului *p*, adresa de început a unei liste simplu înlăntuite. Presupunem că la apel i se transmite adresa primului nod a unei liste în care sunt reținute în ordine valorile întregi -1, 3, 5, 13, 53, 21, 43, 5. Ce valoare va returna funcția în acest caz?

```

function Vm(p:point):integer;
begin
if p^.leg==NULL then Vm:=p^.inf
else
  if (p^.inf>Vm(p^.leg)) then
    Vm:=p^.inf
  else Vm:=Vm(p^.leg)
end;

```

```

int Vm(point *p)
{
  if (p->leg==NULL)
    return p->inf;
  if (p->inf>Vm(p->leg))
    return p->inf;
  return Vm(p->leg);
}

```

a) -1;                  b) 5;

c) 43;                  d) 53.

4. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point^=nod;
  nod=record
    inf : integer;
    leg : point; end;

```

```

struct point
  { int inf;
    point *leg;
  };

```

Funcția *Vm*, declarată în continuare, primește, prin intermediul parametrului *p*, adresa de început a unei liste simplu înlăntuite.

```

function Vm(p:point):boolean;
begin
  if p=nil then Vm:=true
  else
    if (p^.inf>0)and(Vm(p^.leg))
      then Vm:=Vm(p^.leg)
    else Vm:=false;
end;

```

Functia va returna valoarea *True* (pentru varianta Pascal), respectiv 1 (pentru varianta C++) dacă:

- a) lista este vidă;
- b) lista are primul element pozitiv;
- c) lista conține numai valori pozitive;
- d) lista nu conține valori nule.

5. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point=^nod;
  nod=record
    inf : integer;
    leg : point; end;

```

Functia *Sg* primește, prin intermediul parametrului *p*, adresa de început a unei liste simplu înlăntuite.

```

function Sg(p:point):boolean;
begin
  if p^.leg=nil then Sg:=true
  else
    if (p^.inf * p^.leg^.inf<0)
      then Sg:=Sg(p^.leg)
    else Sg:=false;
end;

```

Functia *Sg* returnează valoarea *True* (pentru varianta Pascal) respectiv 1, (pentru varianta C++) dacă:

- a) lista are un singur element;
- b) lista are ultimul element de semn contrar primului;
- c) oricare două elemente succesive din listă au același semn;
- d) oricare două elemente succesive din listă au semne diferite.

6. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point=^nod;
  nod=record
    inf : integer;
    leg : point; end;

```

```

int Vm(point *p)
{
  if (p==NULL)
    return 1;
  if (p->inf>0 && Vm(p->leg))
    return Vm(p->leg);
  return 0;
}

```

Functia *X* primește, prin intermediul parametrilor *p* și *q*, adresele de început ale unor liste simplu înlăntuite.

```

function X(p,q:point):boolean;
begin
  if p=nil then
    if q=nil then X:=true
    else X:=false
  else
    X:=X(p^.leg,q^.leg)
end;

```

```

int X(point *p,point *q)
{
  if (p==NULL) {
    if (q==NULL) return 1;
    return 0;
  }
  return X(p->leg,q->leg);
}

```

Functia *X* returnează valoarea *False* (pentru varianta Pascal), respectiv 0 (pentru varianta C++) dacă:

- a) ambele liste sunt vide;
- b) cele două liste conțin același număr de elemente;
- c) lista a cărei adresă de început este transmisă prin *p* conține mai puține elemente;
- d) lista a cărei adresă de început este transmisă prin *q* conține mai puține elemente.

7. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point=^nod;
  nod=record
    inf : integer;
    leg : point; end;

```

```

struct point
{
  int inf;
  point *leg;
};

```

Functia *Vs* primește, prin intermediul parametrului *p*, adresa de început a unei liste simplu înlăntuite.

```

function Vs(p:point):integer;
begin
  if p=nil then Vs:=0
  else
    if frac(sqrt(p^.inf))=0.0 then
      Vs:=Vs(p^.leg)+p^.inf
    else
      Vs:=Vs(p^.leg)
end;

```

```

int Vs(point *p)
{
  if (p==NULL) return 0;
  if (floor(sqrt(p->inf))==sqrt(p->inf))
    return p->inf+Vs(p->leg);
  return Vs(p->leg);
}

```

Ce rezultat va returna funcția *Vs*, dacă la apel este transmisă adresa de început a unei liste ce memorează valorile 1, 2, 3, 4, 5, 6, 7, 8, 9 ?

- a) 19;              b) 1;              c) 10;              d) 14;              e) 13.

8. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point=^nod;
  nod=record
    inf : integer;
    leg : point; end;

```

```

struct point
{
  int inf;
  point *leg;
};

```

Care dintre următoarele secvențe de instrucțiuni permite crearea unei liste simplu înălțătute nevidă, ale cărei elemente formează un șir de valori întregi consecutive și la cărei adresă de început este memorată de variabila *Prim*?

- a)  

```
new(p); x:=10;
while p<>nil do begin
  p^.inf:=x; x:=x+1; p:=p^.leg
end;
Prim:=p;
```
- b)  

```
read(x); x:=abs(x); Prim:=nil;
while x>0 do begin
  new(p); p^.inf:=x; dec(x);
  p^.leg:=Prim; Prim:=p
end;
```
- c)  

```
x:=30; Prim:=nil;
while x<100 do begin
  p^.inf:=x; x:=x+1;
  p^.leg:=Prim; new(p);
end;
```
- d)  

```
new(p); x:=20; Prim:=nil;
while x<=10 do begin
  p^.inf:=x; x:=x-1; p^.leg:=Prim;
  Prim:=p; new(p);
end;
```

9. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înălțătuite:

```
type point=^nod;
nod=record
  inf : integer;
  leg : point; end;
```

```
struct point
{
  int inf;
  point *leg;
};
```

Care dintre următoarele secvențe de instrucțiuni sunt corecte sintactic, știind că variabila *p* aparține tipului *point*?

- a)  

```
readln(p);
```
- b)  

```
if p<>nil then writeln(p^.leg);
```
- c)  

```
p:=p^.leg^.leg^.inf;
```
- d)  

```
new(p); p^.leg:=p;
```
- a)  

```
cin>>p;
```
- b)  

```
if (p!=NULL) cin>>p->leg;
```
- c)  

```
p=p->leg->leg->inf;
```
- d)  

```
p=new point; p->leg=p;
```

10. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înălțătuite:

```
type pointd=^nod;
nod=record
  inf : integer;
  ls,ld : pointd; end;
```

```
struct pointd
{
  int inf;
  pointd *ls,*ld;
};
```

Care dintre următoarele subprograme modifică adresa de început a listei pe care o primește la apel?

a)

```
procedure Print(Prim:pointd);
begin
  while Prim<>nil do begin
    write(Prim^.inf, ' ');
    Prim:=Prim^.ld
  end
end;
```

b)

```
procedure Print(Prim:pointd);
var p:pointd;
begin
  p:=Prim;
  while P^.ld<>nil do begin
    if Prim^.ld<>nil then
      Prim:=Prim^.ld^.ld;
    p:=p^.ld
  end
end;
```

c)

```
procedure Print(var p:pointd);
begin
  while p<>nil do begin
    write(p^.inf, ' ');
    p:=p^.ld;
  end
end;
```

a)

```
void print(pointd *prim)
{
  for (;prim!=NULL;prim=prim->ld)
    printf("%d ", prim->inf);
}
```

b)

```
void print(pointd *prim)
{
  pointd *p;
  for(p=prim;p->ld;p=p->ld)
    if (prim->ld)
      prim=prim->ld->ld;
}
```

c)

```
void print(pointd *&p)
{
  for (;p!=NULL;p=p->ld)
    printf("%d ", p->inf);
}
```

11. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înălțătuite:

```
type pointd=^nod;
nod=record
  inf : integer;
  ls,ld : pointd; end;
```

```
struct pointd
{
  int inf;
  pointd *ls,*ld;
};
```

Care dintre următoarele subprograme verifică, în mod corect, dacă toate elementele unei liste dublu înălțătuite sunt valori pare?

a)

```
function Ok(p:pointd):boolean;
begin
  if p=nil then Ok:=true
  else
    if odd(p^.inf) then
      Ok:=False
    else Ok:=Ok(p^.ld)
end;
```

b)

```
function Ok(p:pointd):boolean;
begin
  if p=nil then Ok:=true
  else
    if odd(p^.inf) then
      Ok:= Ok(p^.ls)
    else Ok:= Ok(p^.ld)
end;
```

c)

```
function Ok(p:pointd):boolean;
begin
  if p=nil then Ok:=true
  else
    if p^.inf and l=1 then
      Ok:=False
    else Ok:=Ok(p^.ld)
end;
```

d)

```
function Ok(p:pointd):boolean;
begin
  if odd(p^.inf) then
    Ok:=False
  else Ok:=Ok(p^.ld)
end;
```

a)

```
int ok(pointd *p)
{
  if (p==NULL)
    return 1;
  if ((p->inf&2)==1)
    return 0;
  return ok(p->ld);
}
```

b)

```
int ok(pointd *p)
{
  if (p==NULL)
    return 1;
  if (p->inf&2==1)
    return ok(p->ls);
  return ok(p->ld);
}
```

c)

```
int ok(pointd *p)
{
  if (p==NULL) return 1;
  if (p->inf&1)
    return 0;
  return ok(p->ld);
}
```

d)

```
int ok(pointd *p)
{
  if (p->inf&1)
    return 0;
  return ok(p->ld);
}
```

12. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înălungite:

```
type pointd=^nod;
  nod=record
    inf : integer;
    pointd *ls,*ld;
  end;
```

Funcția *Ok* primește, prin intermediul parametrului *p*, adresa celui de al doilea element al unei liste dublu înălungite:

```
function Ok(p:pointd):boolean;
begin
  if p^.ld=nil then Ok:=false
  else
    if p^.ls^.inf=p^.ld^.inf then
      Ok:=True
    else Ok:=Ok(p^.ld)
end;
```

```
int ok(pointd *p)
{
  if (p->ld==NULL)
    return 0;
  if ((p->ls->inf)==(p->ld->inf))
    return 1;
  return ok(p->ld);
}
```

Funcția *Ok* returnează valoarea *True* (pentru varianta Pascal), respectiv 1 (pentru varianta C++) dacă:

- a) oricare două elemente successive din listă au semne diferite;
- b) lista are toate elementele de valori egale;
- c) există un element în listă al cărui vecini au valori egale;
- d) lista conține elemente de valori consecutive.

13. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înălungite:

```
type pointd=^nod;
  nod=record
    inf : integer;
    pointd *ls,*ld;
  end;
```

```
struct pointd
{
  int inf;
  pointd *ls,*ld;
};
```

Considerând că subprogramele următoare primesc la apel adresele de început a două liste dublu înălungite, care dintre ele realizează în mod corect concatenarea celor două liste?

a)

```
procedure C(p1,p2: pointd);
var p: pointd;
begin
  p:=p1;
  while (p^.ld<>nil) do p:=p^.ld;
  p^.ld:=p2;
  p2^.ls:=p;
end;
```

b)

```
procedure C(p1,p2: pointd);
var p: pointd;
begin
  p:=p1;
  while (p<>nil) do p:=p^.ld;
  p^.ld:=p2;
  p2^.ls:=p;
end;
```

c)

```
procedure C(p1,p2: pointd);
var p: pointd;
begin
  p:=p1;
  if p^.ld<>nil then begin
    repeat
      p:=p^.ld;
    until p^.ld=nil;
    p^.ld:=p2;
    p2^.ls:=p;
  end;
end;
```

a)

```
void C(pointd *p1,pointd *p2);
{
  pointd *p;
  for (p=p1;p->ld;p=p->ld);
  p->ld=p2;
  p2->ls=p;
}
```

b)

```
void C(pointd *p1,pointd *p2)
{
  pointd *p;
  for (p=p1;p;p=p->ld);
  p->ld=p2;
  p2->ls=p;
}
```

c)

```
void C(pointd *p1,pointd *p2)
{
  pointd *p;
  p=p1;
  if (p->ld!=NULL)
  {
    do
      p=p->ld;
    while (p->ld!=NULL);
    p->ld=p2;
    p2->ls=p;
  }
}
```

```

4) procedure C(p1,p2: pointd);
var p: pointd;
begin
  p:=p1;
  if p^.ld<>nil then begin
    repeat
      p:=p^.ld;
    until p=nil;
    p^.ld:=p2; p2^.ls:=p;
  end;
end;

```

14. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înlăntuite:

```

type pointd^nod;
nod=record
  inf : integer;
  pointd ^ls,*ld;
  ls,ld : pointd; end;

```

Adresa de început a unei liste dublu înlăntuite care memorează numerele naturale consecutive de la 9 da 1 este reținută în variabila *prim*. Ce se va afișa în urma apelului *print(prim^.ld)* - varianta Pascal, respectiv *print(prim->ld)* varianta C++?

```

procedure print(p:pointd);
begin
  if p^.ld<>nil then begin
    write(p^.ls^.inf,' ');
    print(p^.ld);
    write(p^.ls^.inf,' ');
  end;
end;

```

- a) 9 8 7 6 5 4 3 3 4 5 6 7 8 9;
- b) 9 8 7 6 5 4 3 2 2 3 4 5 6 7 8 9;
- c) 9 8 7 6 5 4 3 4 5 6 7 8 9;
- d) 9 9.

15. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înlăntuite:

```

type pointd^nod;
nod=record
  inf : integer;
  pointd ^ls,*ld;
  ls,ld : pointd; end;

```

Fie două liste dublu înlăntuite. Adresa de început a primei liste este reținută de variabila *prim1*, iar adresa ultimului element din lista a două, de variabila *ultim2*. Știind că cele două liste rețin în ordine valorile 4, 12, 4, 3, 2 respectiv 1, 3, 4, 12, 6 ce valoare va fi afișată în urma apelului *writeln(ver(prim1,ultim2)) / printf("%d\n",ver(prim1,ultim2))*? Definiția funcției *ver* este următoarea:

```

d) void C(pointd *p1,pointd *p2)
{
  pointd *p;
  p=p1;
  if(p->ld!=NULL)
  {
    do p=p->ld; while(p!=NULL);
    p->ld=p2;
    p2->ls=p;
  }
}

```

```

function ver(p,q:pointd):integer;
begin
  if (p<>nil)&(q<>nil) then
    if p^.inf=q^.inf then
      ver:=p^.inf
    else ver:=ver(p^.ld,q^.ls)
    else ver:=0;
  end;

```

a) 4;                    b) 3;

```

int ver(pointd *p, pointd *q) {
  if (p!=NULL&&q!=NULL) {
    if (p->inf==q->inf)
      return p->inf;
    return ver(p->ld,q->ls);
  }
  return 0;
}

```

c) 12;                   d) 0.

16. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point=^nod;
nod=record
  inf : integer;
  point ^leg;
end;

```

```

struct point
{
  int inf;
  point ^leg;
};

```

Identificați care dintre următoarele subprograme verifică, în mod corect, dacă două liste liniare simplu înlăntuite conțin exact aceleași informații. Se consideră că la apel funcțiile primesc, prin intermediul celor doi parametri, adresele de început ale celor două liste:

a) function Ok(p,q:point):boolean;
begin
 if (p<>nil)&(q<>nil) then
 if p^.inf<>q^.inf then
 ok:=false
 else ok:=ok(p^.leg,q^.leg)
 else ok:=true;
 end;

a)

```

int ok(point *p,point *q)
{
  if (p==NULL&&q==NULL)
    return 1;
  if (p!=NULL&&q!=NULL) {
    if (p->inf!=q->inf)
      return 0;
    return ok(p->leg,q->leg);
  }
  return 1;
}

```

b) function Ok(p,q:point):boolean;
begin
 if (p=nil)&(q=nil) then
 Ok:=true
 else
 if (p<>nil)&(q<>nil) then
 if p^.inf<>q^.inf then
 ok:=false
 else ok:=Ok(p^.leg,q^.leg)
 else ok:=false
 end;

b)

```

int ok(point *p,point *q)
{
  if (p==NULL&&q==NULL)
    return 1;
  if (p!=NULL&&q!=NULL)
  {
    if (p->inf!=q->inf)
      return 0;
    return ok(p->leg,q->leg);
  }
  return 0;
}

```

c) function Ok(p,q:point):boolean;
begin
 Ok:=true;
 while (p<>nil)&(q<>nil) do

```

begin
  if p^.inf<>q^.inf then
    Ok:=false;
    p:=p^.leg; q:=q^.leg;
  end;
end;

(d)
function Ok(p,q:point):boolean;
begin
  Ok:=true;
  while (p<>nil)and(q<>nil) do
  begin
    if p^.inf<>q^.inf then Ok:=false;
    p:=p^.leg; q:=q^.leg;
  end;
  if p<>q then Ok:=false;
end;

```

17. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type point=^nod;
nod=record
  inf : integer;
  point *leg;
end;

```

Ce valoare returnează funcția de mai jos în urma apelului  $Nr(prim^.leg)$ , dacă prim reprezintă adresa de început a unei liste care memorează, în ordine, valorile 3, 3, 8, 8, -3, 7, 7, 7?

```

Function Nr(p:point):integer;
var x:integer;
begin
  x:=0;
  while(p^.leg<>nil) do begin
    if (p^.inf=p^.leg^.inf) then inc(x);
    p:=p^.leg;
  end;
  Nr:=x;
end;

```

a) 4; b) 2;

```

int nr(point *p)
{
  int x=0;
  for(;p->leg!=NULL;p=p->leg)
    if (p->inf==p->leg->inf)
      x++;
  return x;
}

```

c) 1; d) 3.

18. Considerăm declarația următorului tip de date necesar reprezentării unei liste simplu înlăntuite:

```

type pointd=^nod;
nod=record
  inf : integer;
  pointd *leg;
end;

```

```

(c)
int ok(point *p,point *q)
{
  for(;p&&q;p=p->leg,q=q->leg)
    if (p->inf!=q->inf)
      return 0;
  return 1;
}

(d)
int ok(point *p,point *q)
{
  for(;p&&q;p=p->leg,q=q->leg)
    if (p->inf!=q->inf)
      return 0;
  return p==q;
}

```

Care dintre următoarele subprograme implementează, în mod corect, operația de căutarea unei valori întregi într-o listă simplu înlăntuită?

a)

```

function ok(x:integer; p:point)
  :boolean;

```

```

var q:point;
begin
  q:=p;
  while q^.inf<>x do q:=q^.leg;
  if q<>nil then ok:=true
  else ok:=false;
end;

```

b)

```

function ok(x:integer; p:point)
  :boolean;

```

```

begin
  ok:=false;
  if p^.inf<>x then
    ok:=ok(x,p^.leg)
  else ok:=true;
end;

```

c)

```

function ok(x:integer; p:point)
  :boolean;

```

```

begin
  while (p<>nil)and(p^.inf<>x)do
    p:=p^.leg;
  ok:=p<>nil;
end;

```

d)

```

function ok(x:integer; p:point)
  :boolean;

```

```

begin
  if p=nil then ok:=false
  else
    if p^.inf<>x then
      ok:=ok(x,p^.leg)
    else ok:=true;
end;

```

a)

```

int ok(int x, point *p)
{

```

```

  point *q;
  for (q=p;q->inf!=x;q=q->leg);
  if (q==NULL)
    return 1;
  return 0;
}

```

b)

```

int ok(int x, point *p)
{

```

```

  if (p->inf!=x)
    return ok(x,p->leg);
  return 1;
}

```

c)

```

int ok(int x, point *p)
{

```

```

  while (p!=NULL&&p->inf!=x)
    p=p->leg;
  return p==NULL;
}

```

d)

```

int ok(int x, point *p)
{

```

```

  if (p==NULL) return 0;
  if (p->inf!=x)
    return ok(x,p->leg);
  return 1;
}

```

```

struct pointd
{
  int inf;
  pointd *ls,*ld;
};

```

19. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înlăntuite:

```

type pointd=^nod;
nod=record
  inf : integer;
  pointd *ls,*ld;
end;

```

Considerăm  $p$  adresa de început a unei liste dublu înălțătute.

- |                              |                             |
|------------------------------|-----------------------------|
| 1) $p^.ld^.ls := q;$         | 1) $p->ld->ls = q;$         |
| 2) $q^.inf := 0;$            | 2) $q->inf = 0;$            |
| 3) $p^.ld := q;$             | 3) $p->ld = q;$             |
| 4) $p^.ld^.ls^.ld := p^.ld;$ | 4) $p->ld->ls->ld = p->ld;$ |
| 5) $\text{new}(q);$          | 5) $q = \text{new pointd};$ |
| 6) $p^.ld^.ls := p;$         | 6) $p->ld->ls = p;$         |

Stabiliti în ce ordine trebuie efectuate instrucțiunile de mai sus astfel încât nodul de la adresa  $q$  să reprezinte al doilea element al listei:

- a) 5, 2, 1, 4, 3, 6;    b) 1, 3, 2, 4, 6, 5;    c) 5, 3, 6, 4, 2, 1;    d) 5, 1, 4, 2, 3, 6.

20. Considerăm declarația următorului tip de date necesar reprezentării unei liste dublu înălțătute:

```
type pointd^=^nod;
nod=record
  inf : integer;
  ls, ld : pointd;
end;
```

```
struct pointd
  { int inf;
    pointd *ls,*ld;
  };
```

Considerăm o listă dublu înălțătă nevidă pentru care variabilele *prim* și *ultim* rețin adresele primului element, respectiv a ultimului element. Considerăm apelul *write(X(prim,ultim))* în Pascal, respectiv *printf("%d", X(prim,ultim))*, unde subprogramul *X* este parțial declarat în continuare:

```
function X(p,u:pointd):integer;
begin
  if (.....) then X := 2
  else
    if (p=u) then X:=1
    else X:=X(p^.ld,u^.ls)+2;
end;

int X(pointd *p,pointd *u)
{
  if (.....) return 2;
  if (p==u) return 1;
  return X(p->ld,u->ls)+2;
}
```

Identificați care dintre expresiile următoare pot înlocui spațiile punctate astfel încât subprogramul *X* să determine numărul de elemente al listei:

- |                     |                     |
|---------------------|---------------------|
| a) $p^.ld = u^.ls$  | a) $p->ld == u->ls$ |
| b) $p^.ld = u$      | b) $p->ld == u$     |
| c) $p = u^.ls$      | c) $p == u->ls$     |
| d) $p^.ld <> u^.ls$ | d) $p->ld != u->ls$ |

## 1.1.2 Probleme rezolvate

1. Într-un fișier text se află scrise, pe o singură linie, numere naturale mai mici sau egale cu 200. Realizați o funcție care creează o listă simplu înălțătă, preluând în ordine valorile din fișier, până la întâlnirea unui număr egal cu precedentul citit. Subprogramul va primi, prin parametrul *s*, un sir de caractere care reprezintă numele fișierului și va returna adresa de început a listei create.  
*Exemplu:* Dacă fișierul text din care se efectuează citirea conține, în ordine, valorile 2 3 1 4 4 6 7 7 8, subprogramul va crea o listă ce conține, în ordine, valorile: 2 3 1 4.

### Soluție:

Crearea listei este realizată prin adăugarea succesivă la finalul acesteia a către unui nou nod (listă tip coadă). Pentru aceasta va fi necesară refinarea succesivă a adresei alocate ultimului nod (*u*). La citirea unei noi valori (*x*) din fișier, se verifică dacă (*u^.inf<>x*) în Pascal, respectiv (*u->inf!=x*) în C++. Adresa alocatea primului nod (*p*) va fi returnată ca rezultat.

```
1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 function creare(s:string):point;
7 var u,q,p:point;
8   x:integer;
9 begin
10   assign(f,s);reset(f);
11   new(p); p^.leg:=nil;
12   read(f,p^.inf);
13   u:=p; read(f,x);
14   while (u^.inf<>x) do begin
15     new(q); q^.leg:=nil;
16     q^.inf:=x; u^.leg:=q;
17     u:=q; read(f,x);
18   end;
19   creare:=p;
20 end;
21 end;
```

```
struct point {
  int inf;
  point *leg;
};

point* creare(char *s)
{
  point *p,*q,*u; int x;
  freopen(s, "r", stdin);
  p=new point;
  p->leg=NULL;
  scanf("%d",&(p->inf));
  u=p; scanf("%d", &x);
  while (u->inf!=x)
  {
    q=new point; q->leg=NULL;
    q->inf=x; u->leg=q;
    u=q; scanf("%d", &x);
  }
  return p;
}
```

2. Realizați un program care afișează numerele obținute prin permutări circulare ale cifrelor unui număr natural *x* preluat de la tastatură. Cifrele lui *x* vor fi reținute într-o listă simplu înălțătă. Vor fi implementate două subprograme:

- subprogramul *Lista*, care permite memorarea cifrelor numărului *x* într-o listă simplu înălțătă. Subprogramul va returna adresa de început a listei create prin intermediul unui parametru;
- subprogramul *Permut*, care primind la apel, prin intermediul unui parametru, adresa de început a unei liste, realizează transferul primului element la finalul listei, fără a folosi memorie suplimentară. Subprogramul va returna noua adresă de început a listei prin intermediul același parametru.

*Exemplu:* Pentru *n*=1234, se va afișa: 2341, 3412, 4123, 1234.

### Soluție:

Vom crea o listă de tip stivă pentru ca ea să conțină cifrele numărului în ordinea scrierii din baza 10. În această situație, adăugarea unui nou nod se face la începutul listei. Adresa alocată primului nod va fi returnată prin parametrul *p*. În cadrul subprogramului *Permut* sunt efectuate următoarele operații:

- parcurgerea listei pentru identificarea adresei ultimului nod (*q*);
- modificarea adresei reținute de *q* în câmpul *leg*;
- actualizarea adresei primului nod al listei;
- memorarea constantei *NIL/NULL* în câmpul *leg* al noului ultim nod al listei.

În varianta Pascal, subprogramele *Lista* și *Permut* sunt implementate ca proceduri.

```

1 type point^=celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5
6 var prim,aux:point;x:longint;
7 procedure Lista(x:longint;
8                   var p:point);
9 var q:point;
10 begin
11   p:=nil;
12   repeat
13     new(q); q^.leg:=p;
14     q^.inf:=x mod 10;
15     x:=x div 10; p:=q;
16   until x=0;
17 end;
18
19 procedure Permut(var p:point);
20 var q:point;
21 begin
22   q:=p;
23   while q^.leg<>nil do
24     q^.leg:=q^.leg;
25   q^.leg:=p; p:=p^.leg;
26   q^.leg^.leg:=nil;
27 end;
28
29 procedure afis(p:point);
30 begin
31   while p<>nil do begin
32     write(p^.inf); p:=p^.leg;
33   end; writeln;
34 end;
35
36 begin
37   readln(x);
38   Lista(x,prim);
39   aux:=prim;
40   repeat
41     permut (prim); afis(prim);
42   until aux=prim;
43 end.

```

```

1 #include <stdio.h>
2
3 struct point {
4   int inf; point *leg;
5 } *prim, *aux; long x;
6
7 void lista(long x,point *&p)
8 {
9   point *q;
10  p=NULL;
11  do {
12    q=new point;
13    q->leg=p;
14    q->inf=x%10;
15    x/=10; p=q;
16  } while (x);
17}
18
19 void permut(point *&p)
20 {
21   point *q;
22   q=p;
23   for(;q->leg!=NULL;q=q->leg);
24   q->leg=p;p=p->leg;
25   q->leg->leg=NULL;
26 }
27
28 void afis(point *p)
29 {
30   for (;p!=NULL;p=p->leg)
31     printf("%d",p->inf);
32   printf("\n");
33 }
34
35 void main()
36 {
37   scanf("%ld",&x);
38   lista(x,prim);
39   aux=prim;
40   do {
41     permut(prim); afis(prim);
42   } while (aux!=prim);
43 }

```

3. Într-o listă simplu înlățuită care memorează întregii ordonați crescător se dorește inserarea unei noi valori astfel încât monotonia valorilor din listă să se păstreze. Realizați un subprogram care efectuează această operație. Atât adresa de început a listei, cât și valoarea de inserat vor fi primite de subprogram la apel, prin intermediul a doi parametri.

### Soluție:

Operația de inserare ca unui nou nod în listă presupune efectuarea a două operații:

- identificarea-căutarea nodului înaintea căruia se va face inserarea în listă;
- alocarea unei adrese pentru noul nod și refacerea legăturilor pentru inserarea acestuia în listă. Dacă noul nod se va insera în fața primului, atunci adresa de început a listei se modifică.

În varianta Pascal, subprogramul *inse* este implementat ca procedură.

```

1 type point^=celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5
6 procedure inse(var p:point;
7                 x:integer);
8 var u,q,t:point;
9 begin
10  q:=p;
11  while (q^.inf<x)and(q<>nil)do
12    begin
13      u:=q;
14      q:=q^.leg;
15    end;
16  new(t);
17  t^.inf:=x;
18  if q=p then begin
19    t^.leg:=p;
20    p:=t;
21  end
22  else begin
23    u^.leg:=t;
24    t^.leg:=q;
25  end;
26 end;

```

4. Considerăm o listă simplu înlățuită în care sunt reținute valori naturale distincte. Să se realizeze un subprogram care șterge elementul din listă ce conține cel mai mare număr prim. Subprogramul va primi la apel, prin intermediul unui parametru, adresa de început a listei.

*Exemplu:* Considerăm lista ce memorează valorile 3, 1, 2, 8, 14, 9, 7, 5. Subprogramul va elimina din listă elementul ce memorează valoarea 7.

Solutie:

- identificarea căutarea nodului ce va fi sters;
- refacerea legăturilor în listă și eliberarea memoriei. Dacă nodul sters este chiar primul, atunci adresa de început a listei se modifică.

Considerăm că există o funcție `Ok` care verifică dacă valoarea naturală primită ca parametru este un număr prim. Ea returnează în Pascal valoarea `True` sau `False`, respectiv 0 sau 1 în C++.

În varianta Pascal, subprogramul *sterge* este implementat ca procedură

```

1 type point^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 procedure sterg(var p:point);
7 var q,t:point; max:longint;
8 begin
9   q:=p; t:=nil;
10  if not ok(p^.inf) then max:=0
11  else max:=p^.inf;
12  while (q^.leg<>nil) do begin
13    if ok(q^.leg^.inf)and
14      (max<q^.leg^.inf)then begin
15      t:=q;
16      max:=q^.leg^.inf;
17    end;
18    q:=q^.leg;
19  end;
20  if max<>0 then
21    if t=nil then begin
22      t:=p;
23      p:=p^.leg;
24      dispose(t);
25    end
26    else begin
27      q:=t^.leg;
28      t^.leg:=t^.leg^.leg;
29      dispose(q);
30    end;
31  end;
32
33  struct point {
34    int inf;
35    point *leg;
36  };
37  .....
38  void sterg(point *&p)
39  {
40    point *q, *t; long max;
41    q=p; t=NULL;
42    if (!ok(p->inf)) max=0;
43    else max=p->inf;
44    for( ; q->leg!=NULL; q=q->leg)
45      if (ok(q->leg->inf) &&
46          max<q->leg->inf)
47      {
48        t=q;
49        max=q->leg->inf;
50      }
51    if (max!=0) {
52      if (t==NULL) {
53        t=p;
54        p=p->leg;
55        / delete t;
56      } else {
57        q=t->leg;
58        t->leg=t->leg->leg;
59        delete q;
60      }
61    }
62  }

```

5. Considerăm o listă simplu înlățuită care reține cifrele unui număr natural. Să se realizeze o funcție care primește la apel, printr-un parametru, adresa de început a unei astfel de liste și returnează cel mai mare număr care se poate forma cu cifrele distincte, memorate în listă.

*Exemplu:* Dacă lista primită conține cifrele 3, 1, 2, 2, 2, 3, 8, 8, 1, 9, 9, 9, atunci subprogramul va returna numărul 98321.

### Solutie:

Functia realizeaza operatia ceruta printr-o parcurgere a listei si o marcare succesiva a cifrelor memorate la fiecare adresă. La finalul parcurgerii, in vectorul *z*, elementul de indice *i* are valoarea *True / 1* (Pascal / C++), daca cifra *i* se gaseste in lista.

```

1 type point=^cellula;
2 cellula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 function nr(p:point):longint;
7 var ap:array[0..9]of boolean;
8   x,i:longint;
9 begin
10   fillchar(ap,sizeof(ap),false);
11   while p<>nil do begin
12     ap[p^.inf]:=true;
13     p:=p^.leg;
14   end;
15   x:=0;
16   for i:=9 downto 0 do
17     if ap[i] then x:=x*10+i;
18   nr:=x;
19 end;

```

```

struct point {
    int inf;
    point *leg;
};

-----
long mr(point *p)
{
    char ap[10];
    long x,i;
    memset(ap,0,sizeof(ap));
    for(;p!=NULL;p=p->leg)
        ap[p->inf]=1;
    x=0;
    for (i=9;i>=0;i--)
        if (ap[i])
            x=x*10+i;
    return x;
}

```

6. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care, primind la apel, prin intermediu unui parametru, adresa de început a unei astfel de liste, separă elementele în două liste, una formată din valorile pare, iar cealaltă, din valorile impare. Subprogramul va returna adresele de început ale celor două liste create prin intermediu a doi parametri.

*Exemplu:* Dacă lista primită conține valorile 3, 1, 2, 2, 4, 7, 8, 1 atunci subprogramul va separa valorile în două liste ce conțin numerele 3, 1, 7, 1, respectiv 2, 2, 4, 8.

Solutie:

Separarea nodurilor în două liste se face simultan cu parcurgerea listei inițiale. Variabilele  $s1$  și  $s2$  reprezintă *santinele* ale listelor nou obținute.

Santinela este un nod fără informație a cărui adresa reprezintă adresa temporară de început a unei liste. Practic, la finalul prelucrării, câmpul *leg* al santinelei memorează adresa reală de început a listei respective.

Pentru a putea realiza separarea nodurilor, se vor reține, în variabilele *u1* și *u2*, adresele ultimelor elemente ale celor două liste. La fiecare pas, un nod al listei inițiale este adăugat fie la finalul listei ce începe de la adresa *s1*, fie la cea care începe de la adresa *s2*.

În varianta Pascal, subprogramul *split* este implementat ca procedură.

```

1 type point^=celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 -----
6 procedure split(p:point;
7           var p1,p2:point);
8 var s1,s2,u1,u2:point;
9 begin
10   new(s1); new(s2);
11   u1:=s1; u2:=s2;
12   while p<>nil do
13     if odd(p^.inf) then begin
14       u1^.leg:=p; u1:=p;
15       p:=p^.leg; u1^.leg:=nil;
16     end
17     else begin
18       u2^.leg:=p; u2:=p;
19       p:=p^.leg; u2^.leg:=nil;
20     end;
21     p1:=s1^.leg; p2:=s2^.leg;
22     dispose(s1); dispose(s2);
23   end;

```

```

struct point {
    int inf;
    point *leg;
};

void split(point *p,
           point *&p1, point *&p2)
{point *s1,*s2,*u1,*u2;
 s1=new point;
 s2=new point;
 u1=s1; u2=s2;
 while ((p!=NULL) {
    if(p->inf&2==1) {
        u1->leg=p; u1=p;
        p=p->leg; u1->leg=NULL;
    } else {
        u2->leg=p; u2=p;
        p=p->leg; u2->leg=NULL;
    }
    p1=s1->leg; p2=s2->leg;
    delete s1;
    delete s2;
}

```

7. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care, primește la apel, prin intermediul unui parametru  $p$ , adresa de început a unei astfel de liste și efectuează stergerea nodurilor de informații negative. Subprogramul va întoarce noua adresă de început a listei tot prin parametrul  $p$ .

*Exemplu:* Dacă lista conține valorile -3, -1, 2, 6, -4, -7, 8, atunci funcția va returna adresa de început a listei ce conține valorile 2, 6, 8.

Solutie:

Stergerea succesivă a nodurilor se va face simultan cu operația de parcurgere a listei. Pentru că este posibil ca primul nod al listei să fie șters în mod repetat, s-a folosit o sentinelă. Aceasta a fost adăugată la începutul listei adică, înaintea nodului de la adresa transmisă la apel prin parametrul  $p$ .

În timpul parcurgerii se păstrează, în variabila  $u$ , adresa nodului situat în listă înaintea nodului curent (memorată adresa  $p$ ). Aceasta este necesară pentru refacerea legăturilor înaintea stergerii propriu-zise (eliberării adresei de memorie respective).

În varianta Pascal, subprogramul *del* este implementat ca procedură.

```
1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 procedure del(var p:point);
7 var s,u,t:point;
8 begin
9   new(s); s^.leg:=p; u:=s;
```

```

struct point {
    int inf;
    point *leg;
};

.....
void del(point *&p)
{
    .....
    point *s,*u,*t;
    s=new point; s->leg=p; u=s

```

```

10 while p<>nil do
11   if p^.inf<0 then begin
12     t:=p; u^.leg:=p^.leg;
13     p:=p^.leg; dispose(t);
14   end
15   else begin
16     u:=u^.leg; p:=p^.leg end;
17   p:=s^.leg; dispose(s);
18 end;

```

18. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care primește la apel, prin intermediul unui parametru, adresa de început a unei astfel de liste și efectuează inserarea celei minime din listă după fiecare element ce conține o valoare pozitivă.

**Exemplu:** Dacă lista conține valorile  $-3, -1, 2, 6, -4, -7, 8$  atunci, după executarea subprogramului, aceasta va conține valorile  $-3, -1, 2, -7, 6, -7, -4, -7, 8, -7$ .

### Solutie.

Funcția *Min* determină valoarea minimă memorată într-o listă și căreia adresa de început o primește prin parametrul *p*.

Inserarea succesivă a unor noi noduri se face simultan cu operația de parcurgere a listei.

În parcursare se verifică dacă elementul curent memorează o valoare pozitivă, caz în care se va insera un nou nod, altfel se trece la nodul următor. După inserarea unui nod, adresa curentă devine adresa nodului situat după cel inserat. În varianta Pascal, subprogramul *ins* este implementat ca procedură.

```

1 type point^=celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 function min(p:point):integer;
7 var m:integer;
8 begin
9   m:=p^.inf;
10  while p<>nil do begin
11    if p^.inf<m then m:=p^.inf;
12    p:=p^.leg;
13  end;
14  min:=m;
15 end;
16 procedure ins(p:point);
17 var q:point; x:integer;
18 begin
19  x:=min(p);
20  while p<>nil do begin
21    if p^.inf>x then begin
22      new(q); q^.inf:=x;
23      q^.leg:=p^.leg;
24      p^.leg:=q; p:=p^.leg^.leg;
25    end
26    else p:=p^.leg
27  end; end;

```

9. Se consideră două liste simplu înlățuită ce conțin valori întregi ordonate crescător. Realizați un subprogram care interclasă cele două liste, fără să folosi memorie suplimentară. Subprogramul va primi la apel, adresele de început a celor două liste prin intermediul a doi parametri și va returna adresa de început a listei obținute prin intermediul celui de al treilea parametru.

*Exemplu:* Dacă cele două liste conțin valorile 2, 4, 6, 8, 10, respectiv 3, 5, 11, 13, subprogramul va returna adresa de început a listei ce conține elementele 2, 3, 4, 5, 6, 8, 10, 11, 13.

#### Soluție:

Parametrii  $p1$  și  $p2$  primesc la apel adresele de început a celor două liste. Subprogramul va returna prin parametrul  $p$ , adresa de început a listei nou obținute. Variabila locală  $u$  va reține adresa ultimului nod adăugat la lista nou obținută. Cele două liste vor fi parcuse simultan, modul curent de informație minimă fiind transferat la finalul listei noi.

În varianta Pascal, subprogramul *incl* este implementat ca procedură.

```

1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 procedure incl(p1,p2:point;
7                 var p:point);
8 var u:point;
9 begin
10   if (p1^.inf<p2^.inf) then
11     begin
12       p:=p1; u:=p1; p1:=p1^.leg
13     end
14   else begin
15     p:=p2; u:=p2;
16     p2:=p2^.leg
17   end;
18   while (p1<>nil)and(p2<>nil)
19   do
20     if p1^.inf<p2^.inf then
21       begin
22         u^.leg:=p1;
23         u:=p1;
24         p1:=p1^.leg;
25       end
26     else begin
27       u^.leg:=p2;
28       u:=p2;
29       p2:=p2^.leg;
30     end;
31   if p1<>nil then u^.leg:=p1
32   else u^.leg:=p2;
33 end;

```

```

1 struct point {
2   int inf;
3   point *leg;
4 };
5 .....
6 void incl(point *p1,
7            point *p2,point *&p)
8 {
9   point *u;
10  if (p1->inf<p2->inf)
11    {p=p1; u=p1;
12     p1=p1->leg;
13    } else
14    {p=p2;
15     u=p2;
16     p2=p2->leg;
17    }
18  while (p1!=NULL&&p2!=NULL)
19    if (p1->inf < p2->inf)
20    {
21      u->leg=p1;
22      u=p1;
23      p1=p1->leg;
24    } else {
25      u->leg=p2;
26      u=p2;
27      p2=p2->leg;
28    }
29  if (p1!=NULL) u->leg=p1;
30  else u->leg=p2;
31 }

```

10. Realizați un subprogram care implementează operația de adunare pe numere mari ( $a+b$ ). Cifrele fiecărui număr vor fi reținute în ordine inversă în cale o listă simplu înlățuită. Subprogramul va primi la apel, prin intermediul parametrilor  $p$  și  $r$ , adresele de început ale celor două liste. Tot prin parametrul  $p$  se va returna adresa listei ce reprezintă suma.

*Exemplu:* Pentru numerele  $a=188$  și  $b=9929$ , cele două liste rețin, în ordine, cifrele 8, 8, 1, respectiv 9, 2, 9, 9. Subprogramul va returna, prin parametrul  $p$ , adresa de început a listei ce conține 7, 1, 1, 0, 1, adică reprezentarea numărului 10117.

#### Soluție:

Subprogramul implementează algoritmul cunoscut de adunare a numerelor mari. Listele sunt parcuse simultan, cifra reținută în prima listă devenind restul modulo 10 a sumei dintre cifrele corespunzătoare din cele două numere și restul anterior( $t$ ).

Dacă o listă s-a epuizat, atunci algoritmul procesează lista care conține încă cifre.

În varianta Pascal, subprogramul *adun* este implementat ca procedură.

```

1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 procedure adun(var p:point;
7                   r:point);
8 var q,q1,u:point;t:integer;
9 begin
10  q:=p; t:=0;
11  while (q<>nil)and(r<>nil)do
12    begin
13      q^.inf:=q^.inf+r^.inf+t;
14      t:=q^.inf div 10;
15      q^.inf:=q^.inf mod 10;
16      u:=q;
17      q:=q^.leg;
18      r:=r^.leg;
19    end;
20  if r<>nil then begin
21    u^.leg:=r; q:=r; end;
22  while (q<>nil) do begin
23    q^.inf:=q^.inf + t;
24    t:=q^.inf div 10;
25    q^.inf:=q^.inf mod 10;
26    u:=q; q:=q^.leg;
27  end;
28  if t<>0 then begin
29    new(q1); q1^.inf:=t;
30    q1^.leg:=nil; u^.leg:=q1;
31  end;
32 end;

```

11. Considerăm o listă circulară în care sunt memorate valori întregi. Să se realizeze o funcție care elimină, începând cu un anumit element, din  $k$  în  $k$  elemente. Operația de stergere se oprește când lista conține un singur element. Valoarea lui  $k$  și adresa elementului de la care se începe stergerea sunt transmise la apel prin doi parametri. Funcția va returna adresa ultimului element rămas în listă.

*Exemplu:* Dacă, începând cu adresa de unde începe stergerea, lista conține valorile 3, 1, 2, 6, 4, 7, 8 și  $k=3$ , atunci funcția va întoarce adresa la care este memorată valoarea 6. S-au eliminat în ordine valorile 2, 7, 1, 8, 4 și 3.

#### Soluție:

Algoritmul parcurge lista și efectuează stergerea până când adresa curentă  $p$  memorează în câmpul  $leg$  adresa  $p$ , deci lista circulară conține un singur element. Stergerea este realizată și cu eliberarea memoriei.

```

1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 function del(var p:point;
7           k:byte):point;
8 var i:integer;t:point;
9 begin
10   while p^.leg<>p do begin
11     for i:=1 to k-2 do
12       p:=p^.leg;
13     t:=p^.leg;
14     p^.leg:=p^.leg^.leg;
15     p:=p^.leg;
16     dispose(t);
17   end;
18   del:=p;
19 end;
20 
```

```

struct point {
    int inf;
    point *leg;
};

.....
point* del(point *&p,
            unsigned char k)
{
    int i; point *t;
    while (p->leg!=p)
    {
        for (i=0;i+2<k;i++) p=p->leg;
        t=p->leg;
        p->leg=p->leg->leg;
        p=p->leg;
        delete t;
    }
    return p;
}

```

12. Realizați un subprogram *Perm* care afișează toate permutările circulare ale unui sir de valori întregi reținute într-o listă circulară. Subprogramul primește, la apel, adresa unui element din listă, considerat ca element de început. El apelează subprogramul *Print* care afișează elementele unei liste circulare, începând de la o anumită adresă primită ca parametru.

*Exemplu:* Pentru lista circulară care conține valorile 1, 2, 3 subprogramul *Perm* va afișa: 1, 2, 3; 2, 3, 1; 3, 1, 2

#### Soluție:

În cadrul subprogramului *perm* se efectuează parcurgerea listei circulare. Adresa fiecărui nod al listei va constitui parametru actual (efectiv) la apelul subprogramului *print*.

Ambele subprograme sunt implementate în Pascal ca proceduri.

```

1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 procedure print(p:point);
7 var t:point;
8 begin
9   t:=p;
10  repeat
11    write(p^.inf,' ');
12    p:=p^.leg;
13  until p=t;
14  writeln;
15 end;
16
17 procedure perm(p:point);
18 var t:point;
19 begin
20   t:=p;
21  repeat
22    print(p);
23    p:=p^.leg;
24  until p=t;
25 end;

```

```

struct point {
    int inf;
    point *leg;
};

void print(point *p)
{
    point *t;
    t=p;
    do {
        printf("%d ", p->inf);
        p=p->leg;
    } while (p!=t);
    printf("\n");
}

void perm(point *p)
{
    point *t;
    t=p;
    do {
        print(p);
        p=p->leg;
    } while (p!=t);
}

```

13. Se consideră un sir de valori reale reținute într-o listă dublu înăntuită. Realizați un subprogram care permite stergerea din listă a elementelor cu număr de ordine par (al doilea, al patrulea, s.a.m.d). Subprogramul va primi, la apel, adresa de început a listei.

*Exemplu:* Lista care retine valorile 1.2, 2.0, 3.2, 4.56, 7.78 va avea, în urma executării subprogramului, valorile 1.2, 3.2, 7.78.

#### Soluție:

Ștergerea succesivă a nodurilor se va face simultan cu operația de parcurgere a listei.

Deoarece lista este dublu înăntuită, la operația de stergere este necesară refacerea legăturilor memorate în ambii vecini ai nodului curent. Ștergerea este realizată cu eliberarea adresei de memorie respective.

În varianta Pascal, subprogramul *del* este implementat ca procedură.

```

1 type pointd=^celula;
2 celula=record
3   inf:double;
4   ls,ld:pointd;
5 end;
6 .....
7 procedure del(p:pointd);
8 var t,q:pointd;
9 begin
10   q:=p^.ld;
11   if p^.ls=&q then
12     p^.ls:=q^.ld;
13   else
14     p^.ls:=q^.ls;
15   dispose(q);
16 end;

```

```

struct pointd
{
    double inf;
    pointd *ls, *ld;
};

.....
void del(pointd *p)
{
    pointd *t,*q;
    q=p->ld;
    if p->ls==q then
        p->ls=q->ld;
    else
        p->ls=q->ls;
    delete q;
}

```

```

11 while q<>nil do begin
12   t:=q;
13   q^.ls^.ld := q^.ld;
14   q^.ld^.ls := q^.ls;
15   if q^.ld<>nil then
16     q:=q^.ld^.ld
17   else q:=nil;
18   dispose(t);
19 end;
20 end;

```

```

while (q!=NULL)
{
    t=q;
    q->ls->ld=q->ld;
    q->ld->ls=q->ls;
    if (q->ld!=NULL)
        q=q->ld->ld;
    else q=NULL;
    delete t;
}

```

14. Realizați un subprogram care inversează legăturile într-o listă simplu înlățuită fără a utiliza memorie suplimentară. În acest fel primul element al listei va deveni ultimul. Subprogramul va primi ca parametru la apel, adresa primului element.

*Exemplu:* Lista care reține valorile 1, 2, 3, 4 va avea, în urma executării subprogramului, valorile 4, 3, 2, 1.

Solutie:

Implementarea recursivă permite memorarea în stivă a nodului următor nodului curent, transmis prin parametrul *p*. Reținerea acestei adrese se face prin intermediu variabilei locale *q*. La revenirea din recursivitate se realizează inversarea legăturilor.

```

1 type point=^celula;
2 celula=record
3   inf:integer; leg:point;
4 end;
5 .....
6 function inv(p:point):point;
7 var q:point;
8 begin
9   if p^.leg=nil then inv:=p
10  else begin
11    q:=p^.leg;
12    inv:=inv(p^.leg);
13    q^.leg:=p;
14    p^.leg:=nil;
15  end;
16 end;

```

```

struct point {
    int inf;
    point *leg;
};

point* inv(point *p)
{
    point *q,*t;
    if (p->leg==NULL)
        return p;
    q=p->leg;
    t=inv(p->leg);
    q->leg=p;
    p->leg=NULL;
    return t;
}

```

15. Considerăm un sir de *n* numere naturale. Pentru fiecare valoare citită se dorește reținerea într-o listă simplu înlățuită a tuturor factorilor primi care apar în descompunerea ei. Realizați un subprogram care permite citirea celor *n* numere și care memorează, într-un tablou unidimensional, fiecare adresă de început a listelor create.

Solutie:

Structura de date folosită permite indexarea adreselor de început a listelor create. Practic, în vectorul *a*, elementul *a[k]* reprezintă adresa de început a listei generate de al *k*-lea număr citit. Dacă această listă este parcursă, atunci pot fi afișați

toți factorii primi care apar în descompunerea celui de-al *k*-lea număr citit. Aceste liste sunt create sub forma unor stive, deoarece fiecare nou factor este plasat, în lista corespunzătoare, înaintea primului nod.

```

1 type point=^celula;
2 celula=record
3   inf:integer;
4   leg:point;
5 end;
6 sir=array[1..100]of point;
7 .....
8 procedure Vec_L(var a:sir);
9 var x,i,j,e:integer;
10 p:point;
11 begin
12   readln(n);
13   for i:=1 to n do begin
14     a[i]:=nil; read(x); j:=2;
15   while x>>1 do begin
16     e:=0;
17     while x mod j=0 do begin
18       inc(e); x:=x div j;
19     end;
20     if e>0 then begin
21       new(p); p^.inf:=j;
22       p^.leg:=a[i]; a[i]:=p;
23     end;
24     inc(j);
25   end;
26 end;
27 end;

```

```

struct point {
    int inf;
    point *leg;
};
typedef point* sir[100];
.....
void vec_l(sir &a)
{
    int x,i,j,e;
    point *p;
    scanf("%d",&n);
    for (i=1; i<=n; i++)
    {
        a[i]=NULL;
        scanf("%d",&x);
        for (j=2; x!=1; j++)
        {
            for (e=0;x%j==0;x/=j)e++;
            if (e>0)
            {
                p=new point;
                p->inf=j; p->leg=a[i];
                a[i]=p;
            }
        }
    }
}

```

### 1.1.3 Probleme propuse

1. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care primește la apel, prin intermediu unui parametru *p*, adresa de început a unei astfel de liste și efectuează transferul primelor două elemente la finalul ei. Nu se va folosi memorie suplimentară. Noua adresă de început a listei va fi returnată de subprogram prin intermediul parametrului *p*.

*Exemplu:* Dacă lista conținea inițial valorile 1, 2, 3, 4, 5, 6, în urma executării subprogramului, ea va reține elementele 3, 4, 5, 6, 1, 2.

2. Considerăm o listă simplu înlățuită ce memorează valori întregi distincte. Realizați o funcție care primește la apel, prin intermediul unui parametru *p*, adresa de început a unei astfel de liste și returnează numărul de elemente memorate în listă înaintea celui de cheie minimă.

*Exemplu:* Dacă lista conține valorile 10, 21, 3, 4, 5, 6, se va afișa 2.

3. Considerăm două liste dublu înlățuită ce memorează valori întregi. Realizați un subprogram care concatenează cele două liste după regula: lista pentru care suma valorilor memorate este mai mare va fi concatenată la finalul celeilalte. Adresele de început ale celor două liste sunt primite de subprogram prin intermediul a doi parametri, iar adresa de început a listei nou obținute va fi returnată prin al treilea parametru.

*Exemplu:* Dacă cele două liste conțin inițial valorile 1, 2, 3, 4, respectiv 5, 6, 10, atunci lista obținută în urma concatenării conține 1, 2, 3, 4, 5, 6, 10.

4. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care primește la apel, prin intermediul unui parametru  $p$ , adresa de început a unei astfel de liste și efectuează inserarea a două elemente de informație nulă înainte și după fiecare element ce reține informația maximă.

*Exemplu:* Dacă lista conține valorile 10, 21, 3, 21, 21, 6, la finalul executării subprogramului lista va memora valorile 10, 0, 21, 0, 3, 0, 21, 0, 0, 21, 0, 6.

5. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care primește la apel, prin intermediul unui parametru  $p$ , adresa de început a unei astfel de liste și efectuează ștergerea elementelor ce rețin ca informație media aritmetică a vecinilor săi.

*Exemplu:* Dacă lista conține valorile 1, 2, 3, 4, 7, 7, 7, la finalul executării subprogramului lista va memora valorile 1, 4, 7, 7.

6. Considerăm o listă dublu înlățuită ce memorează un sir de valori întregi. Realizați un subprogram care primește la apel, prin intermediul parametrilor  $p$  și  $u$ , adresa primului și a ultimului element ale unei astfel de liste și efectuează ștergerea elementului din mijloc. Dacă lista conține un număr par de elemente, atunci se vor sterge cele două elemente situate în mijloc.

*Exemplu:* Dacă lista conține valorile 1, 2, 3, 4, 7, 7, la finalul executării subprogramului lista va memora valorile 1, 2, 7, 7.

7. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care primește la apel, prin intermediul unui parametru  $p$ , adresa de început a unei astfel de liste și efectuează ștergerea elementelor cu număr de ordine par: al doilea, al patrulea, al săselea, și.a.m.d.

*Exemplu:* Dacă lista conține valorile 1, 5, 3, 4, 7, 6, 8, la finalul executării subprogramului lista va memora valorile 1, 3, 7, 8.

8. Considerăm două liste simplu înlățuite care rețin valori întregi ordonate crescător. Realizați un subprogram care interclasează două astfel de liste prin inversarea legăturilor elementelor. Subprogramul primește adresele de început ale listelor prin doi parametri și returnează tot printr-un parametru adresa listei obținute.

*Exemplu:* Dacă cele două liste conțin inițial valorile 1, 2, 8, 9, respectiv 5, 6, 10, atunci lista obținută în urma interclasării conține 1, 2, 5, 6, 8, 9, 10.

9. Considerăm o listă simplu înlățuită ce memorează valori distincte întregi. Realizați un subprogram care primește la apel, prin intermediul unui parametru  $p$ , adresa de început a unei astfel de liste și efectuează ștergerea elementelor situate între cele care rețin valorile  $x$ , respectiv  $y$ . Aceste două valori întregi sunt primite de subprogram tot prin parametri.

*Exemplu:* Considerăm  $x=3$ ,  $y=6$  și lista 1, 5, 6, 4, 7, 3, 8. La finalul executării subprogramului lista va memora valorile 1, 5, 8.

10. Considerăm o listă circulară simplu înlățuită ce memorează cifre din baza 10. Realizați o funcție care primește la apel, prin intermediul unui parametru  $p$ , adresa de început a unei astfel de liste și efectuează ștergerea primului și al ultimului element. Funcția va returna noua adresă de început a listei.

*Exemplu:* Dacă lista conținea cifrele 1, 5, 3, 4, 7, 6, 8, la finalul executării subprogramului lista va memora valorile 5, 3, 4, 7, 6.

11. Fie o listă simplu înlățuită de numere întregi menule. Realizați un subprogram, care primind la apel, prin intermediul unui parametru, adresa de început a unei astfel de liste, permite inserarea între oricare două noduri ce memorează valori de același semn, a unui nou element a cărui valoare este egală cu suma celor două.

*Exemplu:* Dacă lista primită memorează valorile 3, 1, -2, 2, 4, 7, -8, 1, atunci subprogramul va modifica lista după cum urmează 3, 4, 1, -2, 2, 6, 4, 11, 7, -8, 1.

12. Scrieți un subprogram care primește, prin intermediul unui parametru  $p$ , adresa de început a unei liste simplu înlățuite și efectuează ștergerea tuturor nodurilor care conțin valori prime. Noua adresă de început va fi returnată de subprogram tot prin intermediul parametrului  $p$ .

*Exemplu:* Dacă lista primită memorează valorile 3, 1, 12, 2, 4, 7, 8, atunci subprogramul va modifica lista după cum urmează: 1, 12, 4, 8.

13. Se citesc de la tastatură valori naturale până la întâlnirea lui 0. Valorile citite sunt memorate într-o listă liniară simplu înlățuită. Pentru o valoare  $x$  citită, să se separe nodurile în două noi liste, una care să conțină valorile mai mici decât  $x$ , iar cea de a doua, pe cele mai mari decât  $x$ .

*Exemplu:* Dacă  $x=4$ , iar lista inițială conține valorile 3, 7, 1, 2, 2, 4, 8, 0, atunci programul va separa valorile în două liste ce conțin numerele 3, 1, 2, 2, 0, respectiv 7, 4, 8.

14. Să se realizeze un program care creează o listă simplu înlățuită de tip stivă ce memorează primele  $n$  numere palindroame de minimum două cifre și care permite afișarea acestora. Programul va conține următoarele subprograme:

- un subprogram care permite crearea stivei. Subprogramul va primi la apel printr-un parametru întreg valoarea  $n$  și va întoarce prin alt parametru adresa vârfului;

- un subprogram care permite afişarea valorii memorate în vârful stivei. Adresa acesteia este primită printr-un parametru;
- o funcție care efectuează extragerea nodului din vârful stivei și căruia adresa este primită printr-un parametru. Adresa noului vârf este returnată de funcție ca rezultat.

*Exemplu:* Dacă  $n=5$  se va afișa  $55, 44, 33, 22, 11$ .

15. Considerăm că în fișierul *in.txt* se află numere întregi separate în cadrul linilor prin căte un spațiu. Să se creeze o listă simplu înlățuită cu valorile preluate de pe prima linie din *in.txt*. Realizați un program în care sunt definite următoarele subprograme:

- o funcție care permite crearea listei. Funcția va întoarce adresa de început a listei;
- un subprogram care să permită inserarea după elementele cu cheia maximă, a altor două elemente ce memorează primii doi multipli ai acestei chei. Subprogramul va primi printr-un parametru adresa de început a listei;
- un subprogram care să permită ștergerea din listă a tuturor elementelor care memorează cheia minimă a listei. Subprogramul va primi prin parametrul  $p$ , adresa de început a listei și va returna tot prin parametrul  $p$ , noua adresa de început.

*Exemplu:* Considerăm că fișierul *in.txt* conține pe prima linie valorile  $3, 7, 1, 2, 1, 4, 7$ . Programul va crea o listă cu aceste numere. Apelul la procedura de inserare va modifica lista astfel:  $3, 7, 14, 21, 1, 2, 1, 4, 7, 14, 21$ . Apelul la procedura de ștergere va modifica lista inițială astfel:  $3, 7, 2, 4, 7$ .

16. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un subprogram care, primind la apel, prin intermediul unui parametru, adresa de început a unei astfel de liste, separă elementele în două liste, una formată din nodurile cu număr de ordine impar (primul, al treilea, ș.a.m.d), iar cealaltă listă, din nodurile cu număr de ordine par (al doilea, al patrulea, ș.a.m.d). Subprogramul va returna adresele de început ale celor două liste create prin intermediul a doi parametri.

*Exemplu:* Dacă lista primită conține valorile  $3, 1, 2, 2, 4, 7, 8, 1$ , atunci subprogramul va separa nodurile în două liste  $3, 2, 4, 8$ , respectiv  $1, 2, 7, 1$ .

17. Considerăm o listă simplu înlățuită ce memorează valori întregi. Realizați un program care identifică informația cu frecvență maximă de apariție. Dacă există mai multe asemenea valori se va afișa una singură.

Programul va conține următoarele subprograme:

- funcția recursivă *Nr*, care returnează numărul de apariții al unei valori într-o listă simplu înlățuită. Valoarea și adresa de început a listei vor fi primite la apel prin intermediul a doi parametri.

- subprogramul *Print*, care permite afișarea informației cu frecvență maximă de apariție dintr-o listă simplu înlățuită, folosind apeluri la funcția *Nr*. Subprogramul primește adresa de început a listei printr-un parametru.

*Exemplu:* Dacă lista primită conține valorile  $3, 1, 2, 2, 4, 7, 2, 2$  atunci se va afișa valoarea 2 deoarece apare de 4 ori.

18. Realizați un subprogram care să schimbe adresele de legătură ale elementelor unei liste simplu înlățuite astfel încât acestea să fie ordonate crescător după valorile pe care le memorează.

19. Se citește de la tastatură un sir de valori naturale care se termină cu 0 (valoare ce nu aparține sirului). Aceste numere sunt memorate cu ajutorul unei liste simplu înlățuite. Realizați un program care permite ștergerea din listă a oricărora două elemente consecutive, de informații egale. Programul va conține următoarele subprograme:

- funcția *Make*, care permite crearea listei și care returnează adresa de început a acesteia;
- funcția *Del*, care realizează ștergerea nodurilor cu proprietatea specificată în enunț. Adresa de început a listei este primită printr-un parametru. Funcția va returna adresa de început a listei după efectuarea ștergerii.
- subprogramul *Print*, care permite afișarea informațiilor memorate într-o listă simplu înlățuită. Subprogramul primește adresa de început a listei printr-un parametru.

*Exemplu:* Dacă lista primită conține valorile  $3, 3, 3, 2, 4, 7, 2, 2$ , la finalul executării programului se va afișa:  $3, 2, 4, 7$ .

20. Construiți și afișați o listă liniară simplu înlățuită alocată dinamic care să conțină primele  $n$  numere prime. Începând cu cel mai mic număr, să se șteargă elementele numărând din  $k$  în  $k$ . Informațiile elementelor șterse vor fi afișate în ordinea eliminării lor.

*Exemplu:* Pentru  $n=5$  și  $k=3$ , se va afișa:  $2\ 3\ 5\ 7\ 11$ . Numerele în ordinea ștergerii elementelor sunt:  $5\ 2\ 11\ 3\ 7$ .

21. Să se creeze o listă simplu înlățuită care să conțină numere naturale distincte, cu valori mai mici ca 200000. Citirea numerelor se încheie la întâlnirea lui 0, valoare care nu va face parte din listă. Realizați un program care să permită afișarea informațiilor situate între elementul de informație minimă și cel de informație maximă. Programul va conține următoarele subprograme:

- a) funcția *Make*, care permite crearea listei și care returnează adresa de început a acesteia;
- b) funcția *Elem* care permite returnarea adresei la care se găsește elementul de informație minimă sau maximă. Funcția primește prin parametrul *prim* adresa de început a listei și prin parametrul întreg *t* valoarea 1 sau -1, după cum se dorește returnarea adresei minimului, respectiv a maximului.

c) subprogramul *Print*, care permite afișarea informațiilor memorate într-o listă simplu înlățuită situată între două adrese primite prin doi parametri.

*Exemplu:* Dacă lista primită conține valorile 3, 13, 3, 2, 4, 7, 1, 12, la finalul executării programului se va afișa 13, 3, 2, 4, 7, 1.

22. Considerăm un sir de  $n$  numere naturale. Pentru fiecare valoare citită se dorește reținerea într-o listă simplu înlățuită a primilor  $n$  multipli ai săi. Realizați un subprogram care permite citirea celor  $n$  numere și care memorează, într-un tablou unidimensional, fiecare adresă de început a listelor create.

23. Considerăm două liste dublu înlățuite care rețin valori întregi și pentru care se cunosc adresele de început. Realizați un subprogram care verifică dacă lista transmisă prin primul parametru se poate obține din a doua (transmisă prin al doilea parametru) prin eliminarea unora dintre elementele acesteia. În urma executării subprogramului se va afișa pe ecran mesajul *DA* sau *NU*.

*Exemplu:* Dacă prima listă conține valorile 3, 13, 3, 2, 4, 7, iar a doua 3, 2, 7, la finalul executării programului se va afișa *DA*.

24. Să se creeze o listă dublu înlățuită cu caractere citite de la tastatură. Citirea se încheie tastând caracterul '.'. Pornind de la aceasta, să se creeze alte două liste care să conțină: una, literele listei inițiale, iar cealaltă, restul caracterelor. Cele două liste vor fi create prin schimbarea legăturilor listei inițiale (fără a folosi memorie suplimentară).

*Exemplu:* Dacă lista primită conține caracterele 'b', 'c', '\*', '2', 'A', 'W', 'k', '2', atunci programul va separa caracterele în două liste ce rețin 'b', 'c', 'A', 'W', 'k', respectiv '\*', '2', '2'.

25. Să se realizeze câte un subprogram pentru calculul sumei, respectiv a produsului a două polinoame memorate cu ajutorul listelor dublu înlățuite. Un element al listei reține coeficientul și gradul unui monom, respectiv adresele elementelor vecine lui (succesor-predecesor).

*Exemplu:* Presupunând că prima listă memorează polinomul  $X^3+2X+1$ , atunci elementele ei vor reține valorile (1,3) (2,1) (1,0). Dacă a doua listă memorează polinomul  $3X+13$ , atunci elementele ei vor reține valorile (3,1) (13,0). Lista care corespunde sumei celor două polinoame va reține valorile (1,3) (5,1) (14,0).

26. Considerăm o listă simplu înlățuită ce memorează cifrele unui număr. Realizați un subprogram care construiește o listă simplu înlățuită circulară cu cifrele distincte din lista a cărei adresă de început o primește printr-un parametru.

*Exemplu:* Dacă lista primită memorează valorile 3, 1, 3, 2, 4, 7, 1, 2, subprogramul va permite crearea unei liste circulare în care vor fi reținute cifrele 3, 1, 2, 4, 7.

27. Se introduc de la tastatură valori reale până când valoarea citită reprezintă media aritmetică a ultimelor două valori introduse. Realizați un subprogram care permite memorarea acestora într-o listă dublu înlățuită în care valorile sunt reținute în ordine crescătoare. Nu se va efectua operația de sortare a valorilor din listă.

*Exemplu:* Dacă se citesc valorile 2,4, 23,1, 4,5, 39,2, 4,0, 3,0, 3,5, atunci lista va reține, în ordine, valorile 2,4, 3,0, 3,5, 4,0, 4,5, 23,1, 39,2.

28. Considerăm două mulțimi memorate cu ajutorul listelor simplu înlățuite. Să se realizeze câte o funcție pentru fiecare dintre operațiile următoare: intersecția, reuniunea și diferența a două mulțimi. Funcțiile vor întoarce adresa de început ale listelor create. În cadrul acestor subprograme se va apela funcția *Apare*, care verifică dacă o valoare se află memorată într-o listă. Atât valoarea cât și adresa de început a listei vor fi primite de funcție prin intermediul a doi parametri. Funcția *Apare* returnează în Pascal valoarea *True* sau *False*, respectiv 0 sau 1 în C++.

*Exemplu:* Dacă prima listă conține valorile 3, 13, 23, 2, 4, 7, iar a doua 3, 2, 7, 19, 34, atunci vor fi create următoarele liste:

Intersecția: 3, 2, 7; Reuniunea: 3, 13, 23, 2, 4, 7, 19, 34; Diferența: 13, 23, 4.

29. Se citesc mai multe numere care sunt memorate într-o listă circulară simplu înlățuită. Citirea se încheie când este introdus un număr egal cu suma precedentelor două. Să se realizeze un program care afișează valorile din listă în ordinea inversă introducerii. Programul va conține un subprogram recursiv care afișează elementele listei în ordinea cerută.

30. Realizați un program care calculează matricea sumă a două matrice rare memorate prin intermediul listelor simplu înlățuite. Prin matrice rară se înțelege o matrice cu majoritatea elementelor nule. Fiecare element din listă reține linia, coloana și valoarea unui element nenul din matricea respectivă.

## 1.2. Arborescente

### 1.2.1 Concepte teoretice fundamentale

#### Arbore

Arbore  $\Leftrightarrow$  graf neorientat conex și aciclic (fără cicluri).

Fie un graf neorientat  $G$  cu  $n$  vârfuri. Următoarele afirmații sunt echivalente:

$G$  este arbore  $\Leftrightarrow G$  are  $n-1$  muchii și este conex

$G$  este arbore  $\Leftrightarrow G$  are  $n-1$  muchii și nu conține cicluri

$G$  este arbore  $\Leftrightarrow$  oricare pereche de noduri este unită printr-un singur lanț.

Arbore cu rădăcină  $\Leftrightarrow$  graf neorientat conex fără cicluri, în care unul dintre noduri este desemnat ca rădăcină. Nodurile pot fi așezate pe niveluri începând cu rădăcina, care este plasată pe primul nivel.

**Rădăcină**  $\Leftrightarrow$  nod special care generează așezarea unui arbore pe niveluri. Această operație se efectuează în funcție de lungimea lanțurilor prin care celelalte noduri sunt legate de rădăcină. Orice nod al unui arbore poate fi desemnat ca rădăcină.

**Descendent**  $\Leftrightarrow$  într-un arbore cu rădăcină, nodul y este descendental nodului x dacă este situat pe un nivel mai mare (ca număr de ordine) decât nivelul lui x și există un lanț care le unește și nu trece prin rădăcină.

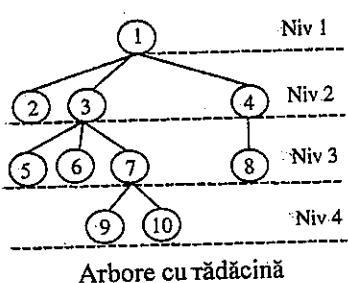
**Descendent direct/fiu**  $\Leftrightarrow$  într-un arbore cu rădăcină, nodul y este fiul (descendantul direct) nodului x dacă este situat pe nivelul imediat următor nivelului lui x și există muchie între x și y.

**Ascendent**  $\Leftrightarrow$  într-un arbore cu rădăcină, nodul x este ascendentul nodului y dacă este situat pe un nivel mai mic decât nivelul lui y și există un lanț care le unește și nu trece prin rădăcină.

**Ascendent direct/părinte**  $\Leftrightarrow$  într-un arbore cu rădăcină, nodul x este părintele (ascendentul direct) nodului y dacă este situat pe nivelul imediat superior (cu număr de ordine mai mic) nivelului lui y și există muchie între x și y.

**Frați**  $\Leftrightarrow$  într-un arbore cu rădăcină, nodul x este frații nodului y dacă au același părinte.

**Frunză**  $\Leftrightarrow$  într-un arbore cu rădăcină, nodul x este frunză dacă nu are nici un descendent direct.



- Nodul 1 este rădăcină.
- Nodurile 5, 6, 7 sunt fiii nodului 3.
- Nodul 7 este părinte pentru 9 și 10.
- Nodul 9 este descendental lui 3
- Nodul 3 este ascendentul lui 10.
- Nodurile 8, 9 și 10 sunt frunze.
- Nodurile 5, 6 și 7 sunt frați.

### Moduri de reprezentare a arborilor

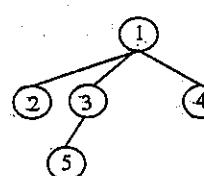
În cazul arborilor, reprezentarea poate fi făcută cu ajutorul:

- matricei de adiacență;
- listei de "descendenți";
- vectorului de tați - numai pentru arborii pentru care s-a desemnat o rădăcină.

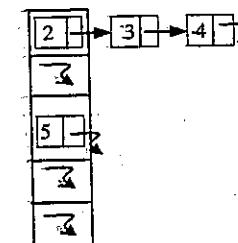
• Lista de descendenți este similară cu lista de adiacență folosită la memorarea grafurilor, pentru fiecare nod reținându-se descendenții direcți. Matricea de adiacență memorează orice graf, deci implicit orice arbore.

• Vectorul de tați memorează pentru fiecare nod, părintele acestuia. Pentru rădăcină, elementul corespunzător din vector este egal cu 0.

Pentru arboarele de mai jos, prezentăm modul de memorare cu ajutorul listelor de descendenți și al vectorului de tați:



$T = (0, 1, 1, 1, 3)$   
Vector de tați (T)



Liste de descendenți (LD)

### Arborescente

**Arborescență**  $\Leftrightarrow$  graf orientat în care sunt îndeplinite simultan condițiile:

- există un singur vârf cu gradul interior nul numit rădăcină;
- toate vâfurile (excepție rădăcina) au gradul interior egal cu 1.

**Descendent**  $\Leftrightarrow$  într-o arborescență, vârful y este descendental vârfului x dacă există drum de la x la y.

**Descendent direct/fiu**  $\Leftrightarrow$  într-o arborescență, vârful y este fiul (descendantul direct) vârfului x dacă există arc de la x la y.

**Ascendent**  $\Leftrightarrow$  într-o arborescență, vârful x este ascendentul vârfului y dacă există drum de la y la x.

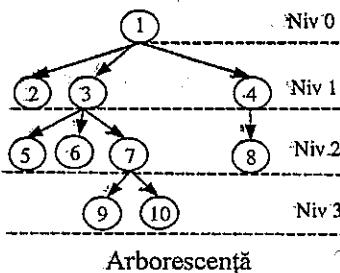
**Ascendent direct/părinte**  $\Leftrightarrow$  într-o arborescență, vârful x este părintele (ascendentul direct) al vârfului y dacă există arc de la x la y.

**Nivelul unui vârf**  $\Leftrightarrow$  într-o arborescență, nivelul vârfului x reprezintă lungimea drumului de la rădăcină la x.

**Frunză**  $\Leftrightarrow$  într-o arborescență, vârful x este frunză dacă are gradul exterior nul.

*Adâncimea unei arborescențe*  $\Leftrightarrow$  lungimea celui mai lung drum de la rădăcină către un alt vârf.

*Subarbore al rădăcinii*  $\Leftrightarrow$  arborescență generată de unul dintre fișii rădăcinii.

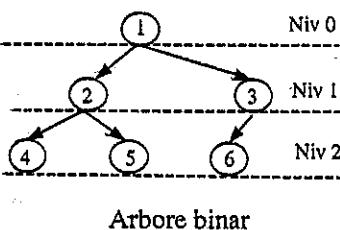


- Vârful 1 este rădăcină.
- Vâfurile 5, 6, 7 sunt fișii vârfului 3.
- Vârful 7 este părinte pentru 9 și 10.
- Vârful 9 este descendental lui 3.
- Vârful 3 este ascendentul lui 10.
- Vâfurile 8, 9 și 10 sunt frunze.
- Adâncimea arborescenței este 3.

*Arbore binar*  $\Leftrightarrow$  arborescență în care fiecare vârf are cel mult doi descendenți direcți denumiți: fiul stâng și fiul drept.

*Arbore binar plin(total)*  $\Leftrightarrow$  arborescență în care fiecare nivel  $i$  este complet, conține  $2^{i-1}$  vârfuri.

*Arbore binar complet*  $\Leftrightarrow$  arborescență în care toate nivelurile sunt complete, exceptie făcând ultimul nivel, unde frunzele sunt plasate de la stânga la dreapta.



- Vârful 1 este rădăcină.
- Arborele binar este complet.
- Vârful 6 este fiul stâng al lui 3.
- Vârful 4 este descendental lui 1.
- Vârful 1 este ascendentul lui 5.
- Vâfurile 4, 5 și 6 sunt frunze.
- Adâncimea arborelui este 2.

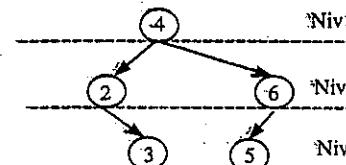
*Arbore de căutare*  $\Leftrightarrow$  arbori binari în care, pentru orice vârf, cheile din subarborele stâng al lui sunt mai mici decât cheia asociată lui, iar cele din subarborele drept sunt mai mari.

*Arbore echilibrați*  $\Leftrightarrow$  arbori binari pentru care diferența absolută dintre adâncimea celor doi subarbore ai oricărui vârf, este cel mult 1.

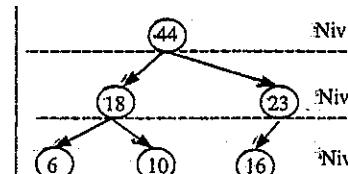
*MaxHeap*  $\Leftrightarrow$  arbore binar complet sau plin în care cheia oricărui vârf este mai mare sau egală cu cheia oricărui descendental său.

*MinHeap*  $\Leftrightarrow$  arbore binar complet sau plin în care cheia oricărui vârf este mai mică sau egală cu cheia oricărui descendental său.

*Arbore AVL*  $\Leftrightarrow$  arbore binar de căutare echilibrat.



Arbore binar de căutare



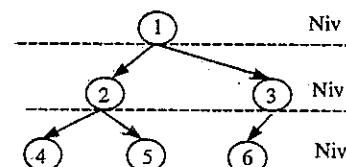
MaxHeap

### Moduri de reprezentare ale arborilor binari

În cazul arborilor binari, reprezentarea poate fi făcută astfel:

- folosind reprezentarea cu ajutorul parantezelor;
- folosind reprezentarea statică standard - cu ajutorul a doi vectori;
- folosind vectorul de 'tați' - în cazul arborilor compleți;
- folosind alocarea dinamică a memoriei.

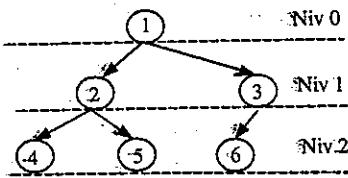
• Reprezentarea cu ajutorul parantezelor se face folosind o expresie algebraică, în care informațiile referitoare fiecarui fiu al unui vârf sunt cuprinse într-o pereche de paranteze rotunde și separate prin virgulă. Lipsa unuia din fii poate fi descrisă printr-un simbol asociat special 'α'. Fiecare paranteză deschisă înseamnă coborârea pe un nou nivel, iar cea închisă, urcarea pe nivelul anterior.



Reprezentarea cu ajutorul parantezelor este următoarea:

1(2(4, 5), 3(6, α))

• Reprezentarea standard este cea în care pentru fiecare vârf se precizează descendental stâng și cel drept. Aceste informații sunt reținute în doi vectori  $St$  și  $Dr$ . Astfel, elementul  $St[i]$  memorează cheia vârfului stâng, iar elementul  $Dr[i]$  pe cea a fiului drept. Lipsa fiului stâng sau drept se reprezintă, în general, prin valoarea 0. O altă variabilă va reține rădăcina arborescenței.



Reprezentarea cu ajutorul vectorilor  $S_t$  și  $D_r$  este următoarea:

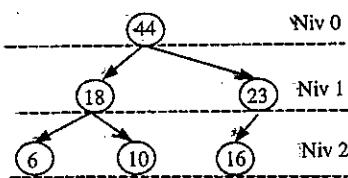
$$S_t = (2, 4, 6, 0, 0, 0)$$

$$D_r = (3, 5, 0, 0, 0, 0)$$

$$rad = 1$$

• Reprezentarea cu ajutorul vectorilor de tați folosită în cazul arborilor compleți pleacă de la presupunerea că numerotarea vârfurilor se face de la 1 la  $n$ , începând cu rădăcina și de la stânga la dreapta, în cadrul fiecărui nivel (vezi figura de mai sus). Astfel pentru vârful  $i$ , elementul  $T[i]$  va reține cheia asociată acestuia. În plus, elementele  $T[2*i]$  și  $T[2*i+1]$  memorează cheile fiului stâng, respectiv cea a fiului drept, pentru orice  $1 \leq i \leq [n/2]$ .

Acest mod de memorare este folosit și în cazul arborescențelor minheap și maxheap. Un astfel de exemplu este prezentat în continuare:



Reprezentarea cu ajutorul vectorului  $T$  a maxheap-ului din figură este:

$$T = (44, 18, 23, 6, 10, 23).$$

Cheia fiului drept al nodului 2 este memorată de  $T[2*2+1] = 10$ .

• Reprezentarea în memoria heap a arborescențelor se face în general în cazul arborilor binari oarecare sau a arborilor binari de căutare.

În acest caz este utilizată o structură dinamică de date în cadrul căreia pentru fiecare vârf se va reține, cu ajutorul unei înregistrări, cheia asociată și adresele de memorie unde se găsesc memorării cei doi fii ai săi. În cazul în care nu există vreunul din fii, se memorează adresa nulă (NIL/NULL).

În limbajul de programare, definiția de tip a unei astfel de structuri poate fi făcută astfel:

```
type varf^=nod;
nod=record
  inf : integer;
  fs, fd : varf; end;
```

```
struct varf
{ int inf;
  varf *fs,*fd;
};
```

## 1.2.2 Teste cu alegere multiplă și duală

1. Considerăm următoarele tablouri unidimensionale. Care dintre ele pot reprezenta vectorul de tați al unui arbore cu 6 noduri?

- a)  $T = (0, 1, 2, 0, 4, 5);$   
b)  $T = (0, 1, 1, 1, 1, 1);$   
c)  $T = (0, 1, 2, 3, 4, 5);$   
d)  $T = (1, 3, 2, 6, 4, 5).$

2. Considerăm următoarele tablouri unidimensionale reprezentând vectorul de tați al unui arbore cu 6 noduri. Care dintre ele codifică un arbore cu 3 frunze?

- a)  $T = (0, 1, 2, 2, 4, 1);$   
b)  $T = (0, 1, 1, 1, 1, 1);$   
c)  $T = (0, 1, 2, 3, 4, 1);$   
d)  $T = (2, 4, 4, 0, 1, 1).$

3. Se consideră un arbore cu rădăcină memorat cu ajutorul vectorului de tați următor:  $T = (2, 3, 0, 3, 3, 2, 6, 6, 4, 8)$ . Identificați care dintre nodurile următoare sunt situate pe niveluri pare (se consideră că rădăcina este situată pe nivelul 1):

- a) 2 4 10 5;  
b) 3 1 6 9;  
c) 2 4 5 7 8;  
d) 1 6 9 10 3.

4. În care dintre arborii următori, nodurile 6, 4 și 8 sunt descendenți ai nodului 3? Arbořii au fost memorati cu ajutorul vectorilor "de tați":

- a)  $T = (3, 0, 2, 3, 2, 3, 4, 4, 3);$   
b)  $T = (3, 3, 4, 0, 2, 3, 4, 4, 4);$   
c)  $T = (0, 3, 1, 3, 2, 3, 4, 4, 3);$   
d)  $T = (9, 9, 4, 9, 9, 9, 9, 9, 0).$

5. Considerând arboarele memorat prin vectorul de tați  $T = (3, 3, 4, 0, 2, 3, 4, 4, 4)$ , care sunt perechile de noduri unite printr-un lanț de lungime maximă?

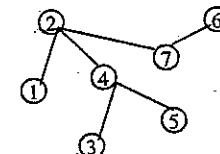
- a) 4 și 5;      b) 5 și 9;      c) 5 și 8;      d) 6 și 7.

6. Considerăm arborele memorat prin vectorul de tați  $T = (2, 0, 2, 1, 4, 1, 6, 7, 4)$ , și nodurile 5 și 8. Identificați care este cel mai apropiat «strâmoș comun» al lor (ascendent pentru ambele noduri situat pe lanțul care le unește).

- a) 4;      b) 7;      c) 6;      d) 1.

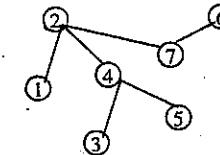
7. Considerând arboarele din figura alăturată, desemnați un nod ca rădăcină, astfel încât numărul de niveluri pe care poate fi așezat să fie maxim.

- a) 1;      b) 4;      c) 6;      d) 2.

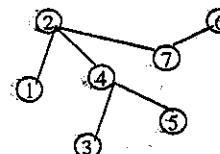


8. Considerând arboarele din figura alăturată, desemnați un nod ca rădăcină, astfel încât numărul de frunze să fie maxim.

- a) 2;      b) 4;      c) 6;      d) 1.



9. Considerând arborele din figura alăturată, desemnați un nod ca rădăcină, astfel încât nodul 4 să aibă exact 3 ascendenți.



- a) 2;      b) 4;      c) 1;      d) 6.

10. Considerăm arborele memorat cu ajutorul vectorului de tați următor  $T = (2, 7, 4, 2, 4, 0, 6)$ . Care dintre următoarele noduri sunt unite de rădăcină prin lanțuri de lungime 4?

- a) 1 și 3;      b) 1 și 4;      c) 5 și 3;      d) 1 și 5.

11. Considerăm un arbore binar reprezentat prin vectorii  $St=(2,4,0,5,0,0)$ ,  $Dr=(3,0,0,6,0,0)$ , în care rădăcina este vârful 1. Care dintre următoarele afirmații sunt adevărate?

- a) Arborele binar este complet;  
b) Numărul frunzelor este egal cu jumătate din numărul vârfurilor;  
c) Vârful 5 are 3 ascendenți;  
d) Vârful 6 este fiul vârfului 2.

12. Considerând arborele binar din figura alăturată, determinați numărul minim de vârfuri ce trebuie adăugate pentru ca arborele să devină complet.

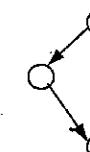
- a) 2;      b) 4;      c) 1;      d) 0.

13. Într-un arbore binar cu 9 vârfuri, în care toate vâfurile ce nu sunt frunze au exact 2 fi, care dintre următoarele afirmații sunt adevărate?

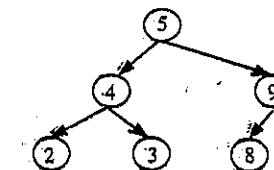
- a) Arborele binar întotdeauna complet;  
b) Numărul frunzelor este egal cu 5;  
c) Numărul frunzelor este egal cu 4 sau cu 5;  
d) Adâncimea maximă a arborelui este 3.

14. Considerând arborele binar din figura alăturată, determinați numărul minim de vârfuri ce trebuie adăugate pentru ca arborele să devină echilibrat.

- a) 2;      b) 4;      c) 1;      d) 3.



15. Considerând arborele binar de căutare din figura alăturată, determinați ordinea în care au fost introduse valorile memorate de el:



- a) 5,4,2,9,3,8;      b) 5,4,8,9,2,3;      c) 5,9,4,3,2,8;      d) 5,2,3,4,8,9.

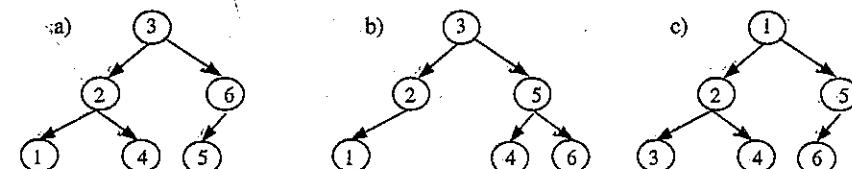
16. Care dintre vectorii următori pot reprezenta vectorul de tați necesar memorării unui arbore binar cu 6 vârfuri?

- a)  $T=(0,1,1,2,2,3)$ ;      b)  $T=(1,2,3,4,5,6)$ ;      c)  $T=(0,1,1,2,2,2)$ ;      d)  $T=(0,1,2,3,4,5)$ .

17. Considerăm ordinea parcurgerii în preordine și postordine a unui arbore binar: preordine 1,2,3,4,5 și postordine 2,4,5,3,1. Identificați căruia dintre arborii următori îi corespund aceste parcurgeri:

- |  |  |  |  |
|--|--|--|--|
| a) $St=(2,0,4,0,0)$<br>$Dr=(3,0,5,0,0)$<br>$rad=1$ | b) $St=(2,0,0,0,0)$<br>$Dr=(3,4,5,0,0)$<br>$rad=1$ | c) $St=(2,4,0,0,0)$<br>$Dr=(3,5,0,0,0)$<br>$rad=1$ | d) $St=(2,3,4,0,0)$<br>$Dr=(0,5,0,0,0)$<br>$rad=1$ |
|--|--|--|--|

18. Identificați care dintre arborii de căutare următori a fost creat la introducerea în ordine a valorilor: 3, 5, 6, 2, 4, 1:



19. Identificați forma parantezată de reprezentare a arborelui binar următor:  $rad=1$ ,  $St=(2,4,0,0,0,0)$ ,  $Dr=(3,5,0,6,7,0,0)$ :

- a)  $1(2(4(\alpha,5), \alpha(\alpha,7)),3)$ ;  
b)  $1(2(4(\alpha,6),5(\alpha,7)),3)$ ;  
c)  $1(2(4(\alpha,6),5(\alpha,3)),7)$ ;  
d)  $1(2(4(\alpha,\alpha),5(6,7)),3)$ .

20. Adâncimea maximă a unui arbore binar cu  $n$  vârfuri este:

- a)  $\log_2 n$ ;      b)  $n$ ;      c)  $n-1$ ;      d)  $\log_2 n + 1$ .

### 1.2.3 Probleme rezolvate

1. Realizați o funcție care permite memorarea în *heap* a unui arbore binar, ale căruia chei (asociate vârfurilor) sunt introduse în preordine. Lipsa unui fiu se semnalizează prin introducerea valorii 0. Funcția va returna adresa rădăcinii arborelui creat.

#### Soluție:

Problema creării unui arbore binar se rezolvă prin metoda *divide et impera*. Astfel, aceasta se va împărti în trei subprobleme independente:

- crearea vârfului curent - considerat de fiecare dată rădăcină (subproblemă de dimensiune 1);
- crearea subarborelui stâng al vârfului curent;
- crearea subarborelui drept al vârfului curent.

Descompunerea ia sfârșit când subproblema creării unui subarbore se referă la un subarbore vid.

În momentul în care ambele subarbore sunt construși, urmează faza de combinare a soluțiilor, în care se unesc subarborele vârfului considerat rădăcina lor.

Implementarea funcției care realizează memorarea în *heap* a unui arbore binar folosește următoarele declarații:

```
type varf = ^nod;
nod = record
  inf : byte;
  fs,fd : varf;
end;

struct varf {
  int inf;
  varf *fs,*fd;
};
```

Variabila locală *p* reprezintă pointer-ul la adresa căruia se va memora vârful curent. La finalul executării, funcția *Tree* va returna adresa rădăcinii arborelui binar creat.

```
function Tree : varf;
var p : varf; x : byte;
begin
  read(x);
  if x = 0 then Tree := nil
  else begin
    new(p); p^.inf := x;
    p^.fs := Tree;
    p^.fd := Tree;
    Tree := p;
  end; end;
}

varf* tree()
{
  varf *p; int x;
  scanf("%d",&x);
  if (x==0) return NULL;
  p=new varf;
  p->inf=x;
  p->fs=tree();
  p->fd=tree();
  return p;
}
```

2. Realizați câte un subprogram care permite traversarea în preordine, inordine și postordine a unui arbore binar alocat în *heap*.

#### Soluție:

Traversarea unui arbore binar constă în parcurgerea succesivă a vârfurilor din arbore și eventual prelucrarea cheilor asociate lor. Pe parcursul traversării, fiecare

vârf este vizitat o singură dată. Există trei modalități fundamentale de traversare a arborilor binari, care diferă una de cealaltă doar prin momentul în care se vizitează rădăcina:

- *traversarea în preordine* – această parcurgere se face prin vizitarea rădăcinii, apoi a subarborelui stâng și în cele din urmă, a subarborelui drept. Modul de vizitare a subarborelor respectă aceeași regulă.
- *traversarea în inordine* – această parcurgere se face prin vizitarea subarborelui stâng, apoi a rădăcinii și în cele din urmă, a subarborelui drept. Modul de vizitare a subarborelor respectă aceeași regulă.
- *traversarea în postordine* – această parcurgere se face prin vizitarea subarborelui stâng, apoi a subarborelui drept și în cele din urmă, a rădăcinii. Modul de vizitare a subarborelor respectă aceeași regulă.

Problema traversării unui arbore binar se rezolvă prin metoda *divide et impera*. Astfel, aceasta se va împărti în trei subprobleme independente:

- vizitarea rădăcinii (subproblemă de dimensiune 1);
- traversarea subarborelui stâng al rădăcinii;
- traversarea subarborelui drept al rădăcinii.

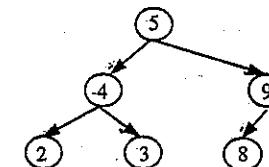
Descompunerea ia sfârșit când subproblema traversării subarborelui se referă la un subarbore vid.

Să luăm ca exemplu arborele alăturat. Ordinea de traversare a vârfurilor este următoarea:

*Preordine* = (5,4,2,3,9,8)

*Inordine* = (2,4,3,5,8,9)

*Postordine* = (2,3,4,8,9,5)



Implementarea subprogramelor care realizează traversarea în preordine, inordine și postordine a unui arbore binar folosește declarațiile de la problema anterioară:

```
procedure PreO(p:varf);
begin
  if p <> nil then begin
    write(p^.inf);
    PreO(p^.fs); PreO(p^.fd);
  end;
end;

procedure InO(p:varf);
begin
  if p <> nil then begin
    InO(p^.fs);
    write(p^.inf);
    InO(p^.fd);
  end;
end;
```

```
void PreO(varf *p)
{
  if (p==NULL) return;
  printf("%d ", p->inf);
  PreO(p->fs);
  PreO(p->fd);
}

void InO(varf *p)
{
  if (p==NULL) return;
  InO(p->fs);
  printf("%d ", p->inf);
  InO(p->fd);
}
```

```

17 procedure PostO(p:varf);
18 begin
19   if p <> nil then begin
20     PostO(p^.fs); PostO(p^.fd);
21     write(p^.inf);
22   end;
23 end;

```

3. Realizați o funcție care permite memorarea în *heap* a unui arbore binar în care fiecare vârf are proprietatea următoare: diferența absolută între numărul de vârfuri din subarborele stâng și cel drept este cel mult 1. Cheile vârfurilor sunt introduse în preordine. Funcția va returna adresa rădăcinii arborelui creat.

#### Solutie:

Condiția ce rezultă din definiția arborelui prezentată în enunț se poate exprima în felul următor: cele  $n$  vârfuri trebuie distribuite în număr de  $[n/2]$  în subarborele stâng și  $[(n-1)/2]$  în subarborele drept.

Funcția *Tree*, va primi prin parametrul  $n$ , numărul de vârfuri din arbore și va returna adresa rădăcinii. Implementarea funcției folosește declarațiile de la problemele anterioare:

```

1 function Tree(n:byte):varf;
2 var p:varf;x:byte;
3 begin
4   new(p); read(p^.inf);
5   Tree := p;
6   if n div 2>0 then
7     p^.fs := Tree(n div 2)
8   else p^.fs := nil;
9   if (n-1)div 2>0 then
10    p^.fd := Tree((n-1)div 2)
11  else p^.fd := nil;
12 end;

```

4. Realizați o funcție care permite inserarea unei noi valori întregi într-un arbore binar de căutare memorat în *heap*. Funcția va primi adresa rădăcinii arborelui de căutare.

#### Solutie:

Operația de inserare a unei noi valori, într-un arbore binar de căutare, presupune o comparare succesivă a cheii vârfului curent cu această valoare.

- dacă valoarea este egală cu cheia atunci nu se efectuează inserarea;
- dacă valoarea este mai mică decât cheia, atunci se încearcă inserarea în subarborele stâng;
- dacă valoarea este mai mare decât cheia, atunci se încearcă inserarea în subarborele drept.

Inserarea propriu-zisă se efectuează când subarborele curent este vid.

```

void PostO(varf *p)
{
  if (p==NULL) return;
  PostO(p->fs);
  PostO(p->fd);
  printf("%d ", p->inf);
}

```

Implementarea subprogramului *InsT*, care realizează inserarea unei valori într-un arbore de căutare, folosește declarațiile de la problemele anterioare:

```

1 procedure InsT(var p:varf;
2                 k:byte);
3 begin
4   if p = nil then begin
5     new(p); p^.inf := k;
6     p^.fs := nil;
7     p^.fd := nil;
8   end
9   else
10  if k < p^.inf then
11    InsT(p^.fs,k)
12  else
13    if k > p^.inf then
14      InsT(p^.fd,k)
15 end;

```

```

void InsT(varf *&p, int k)
{
  if (p==NULL)
  {
    p=new varf;
    p->inf=k;
    p->fs=p->fd=NULL;
  }
  else
  if (k<p->inf) InsT(p->fs,k);
  else
  if (k>p->inf) InsT(p->fd,k);
}

```

5. Realizați un subprogram care permite ștergerea unui vârf de cheie dată, dintr-un arbore de căutare. Subprogramul va primi la apel, prin parametrul  $x$ , cheia vârfului ce trebuie șters și, prin parametrul referință  $p$ , adresa rădăcinii.

#### Solutie:

Pentru a șterge un vârf dintr-un arbore de căutare, trebuie mai întâi localizat. Există patru modalități ce intervin la operația efectivă de ștergere:

- vârful este frunză, caz în care părintele lui va memora adresa *nullă* în locația corespunzătoare adresei fiului;
- vârful are subarborele drept vid, caz în care subarborele lui stâng va fi atașat ca subarbore stâng al părintelui său;
- vârful are subarborele stâng vid, caz în care subarborele lui drept va fi atașat ca subarbore drept al părintelui său;
- vârful are ambeii subarbore nevidi. În acest caz, se va identifica cea mai mare cheie ce se găsește în subarborele lui stâng. Această valoare maximă va înlocui cheia vârfului ce trebuia șters. După această mutare, se va șterge vârful ce reținea cheia maximă. El are însă doar un singur subarbore nevid: cel stâng.

Subprogramul *MaxS* primește, la apel, prin intermediul parametrului  $p$ , adresa vârfului care trebuie șters iar, prin parametrul  $t$ , adresa fiului său stâng. Subprogramul identifică vârful de cheie maximă din subarborele stâng al lui  $p$  și efectuează ștergerea acestuia, numai după mutarea cheii sale în vârful  $p$ .

Subprogramul *Del* efectuează ștergerea unui vârf dintr-un arbore de căutare, în toate cele patru situații.

Implementarea ambelor subprograme ia în considerare declarațiile de la problemele anterioare:

```

1 procedure MaxS(var p,t:varf);
2 var x:varf;
3 begin
4 if t^.fd = nil then begin
5   p^.inf := t^.inf; x := t;
6   t := t^.fs; end
7 else MaxS(p,t^.fd)
8 end;
9
10 procedure Del(var p:varf; k:byte);
11 var x,t:varf;
12 begin
13 if p^.inf>k then Del(p^.fs,k)
14 else if p^.inf< k then Del(p^.fd,k)
15 else
16 if (p^.fs=nil)and(p^.fd=nil)
17 then p := nil
18 else
19 if (p^.fd = nil)then begin
20   t := p^.fd; dispose(p);
21   p := t;
22 end else
23 if (p^.fs = nil)then begin
24   t := p^.fs; dispose(p);
25   p := t;
26 end else MaxS(p,p^.fs)
27 end;

```

6. Realizați un subprogram care verifică dacă doi arbori binari au structură identică. Subprogramul va primi la apel adresele celor două rădăcini și va returna, prin intermediul parametrului *ok*, valoarea 1 dacă arborii sunt identici sau 0, în caz contrar.

#### Soluție:

Subprogramul traversează simultan, în preordine, cele două arbori binari. La întâlnirea situației în care un subarbore este vid iar, celălalt nu, parametrul referință *ok* își modifică valoarea în 0.

Implementarea subprogramului *equal*, ia în considerare declarațiile de la problemele anterioare:

```

1 procedure equal(p,q:varf;
2   var ok:byte);
3 begin
4 if (p = nil)<>(q = nil) then
5   ok := 0
6 else
7 if (p<>nil)and(q<>nil)then
8 begin
9   equal(p^.fs, q^.fs, ok);
10  equal(p^.fd, q^.fd, ok);
11 end
12 end;

```

```

void MaxS(varf *&p,varf *t)
{
varf *x;
if (t->fd==NULL)
{
  p->inf=t->inf; x=t;
  t=t->fs;
} else MaxS(p,t->fd);
}

void Del(varf *&p,int k)
{
varf *t;
if (p->inf>k) Del(p->fs,k);
else
  if (p->inf<k) Del(p->fd,k);
else
  if (p->fs==NULL&&p->fd==NULL)
    p = NULL;
else
  if (p->fs==NULL) {
    t = p->fd; delete p; p = t; }
  else
    if (p->fd==NULL) {
      t = p->fs; delete p; p = t; }
  else MaxS(p,p->fs);
}

```

```

13 begin
14   ... ok := 1;
15   if equal(root1, root2, ok);
16   writeln(ok);
17 end.

```

```

void main()
{
... ok=1;
if equal(root1,root2,ok);
printf("%d\n", ok);
}

```

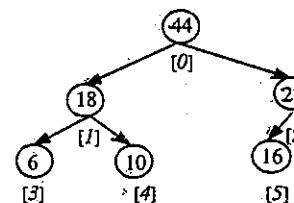
7. Realizați un subprogram care permite inserarea unei noi valori într-un *maxheap* memorat în vectorul *H*. Subprogramul primește prin intermediul unui parametru *k* poziția finală din vector, pe care a fost inserată această nouă valoare în *maxheap*.

*Exemplu:* Pentru *H*=(44,18,23,6,10,16,52) se va afisa *H*=(52,18,44,6,10,16,23).

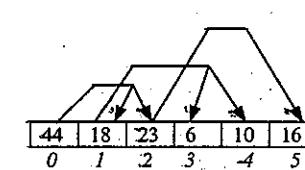
#### Soluție:

Presupunem că heap-ul conține *n* elemente și este memorat în vectorul *H*, începând cu indicele 0. În acest caz, pentru vârful a cărui cheie este memorată în *H[i]*, cheia fiului stâng este *H[2\*i+1]*, iar cheia fiului drept este *H[2\*i+2]*.

Invers, părintele vârfului a cărui cheie este memorată în *H[i]* se regăsește în vector pe poziția  $\lceil (i-1)/2 \rceil$  (în varianta Pascal,  $(i-1) \text{ div } 2$ ).



MaxHeap în reprezentare logică



MaxHeap memorat în vector

Problema se rezolvă printr-o parcurgere a maxheap-ului, prin tați, către rădăcină. Practic, ultimul element inserat în heap se urcă în arbore pe poziția corectă, prin interschimbări succesive cu părintele său. Algoritmul ia sfârșit când cheia inserată a ajuns într-un vârf al cărui părinte are o cheie mai mare.

Subprogramul *HeapUp* realizează inserarea unei valori plasate inițial pe ultima poziție într-un heap. Implementarea urmărește algoritmul descris anterior. Se iau în considerare următoarele declarații:

```

type sir=array[0..100]of byte;
var H:sir;
  i,j,n:byte;

```

```

typedef int sir[101];
sir H; int i, j, n;

```

Subprogramul *swap* realizează interschimbarea a două elemente din *maxheap-ul* *H*, situate pe indicii transmiși prin parametrii *k* și *t*.

```

procedure swap(k,t:byte);
var x:byte;
begin
  x:=H[t]; H[t]:=H[k]; H[k]:=x;
end;

```

```

void swap(int k, int t)
{
  int x;
  x=H[t]; H[t]=H[k]; H[k]=x;
}

```

```

6 procedure HeapUp(k:byte);
7 var t:byte;
8 begin
9   if k > 0 then begin
10     t := (k-1) div 2;
11     if H[k] > H[t] then begin
12       swap(k,t);
13       HeapUp(t);
14     end;
15   end;
16 end;

```

8. Realizați un subprogram care construiește un maxheap din elementele unui vector, aflate într-o ordine oarecare. La apel, subprogramul primește prin parametrul  $k$  indicele maxim din vector.

*Exemplu:* Pentru  $H=(23,44,15,16,18,10,6)$ , se va afișa  $H=(44,23,15,16,18,10,6)$ .

#### Soluție:

Algoritmul care rezolvă problema trebuie să permită reînajarea elementelor în vector, astfel încât, pentru fiecare vârf corespunzător din heap, ambii fi și rețină cheia de informație mai mică. Această operație se realizează considerând vectorul împărțit în două părți: prima conținând elementele unui maxheap, iar ultima valori ce vor fi introduse, pe rând, în maxheap. Inițial, prima parte conține un maxheap format dintr-un singur vârf.

La fiecare iterație a algoritmului, una din cele  $n-1$  valori va fi introdusă în maxheap. Această inserare se va realiza folosind subprogramul *HeapUp* prezentat în problema anterioară.

Implementarea subprogramului *BuildH*, care construiește un maxheap conform strategiei prezentate, ia în considerare următoarele declarații:

```

type sir=array[0..100]of byte;    typedef int sir[101];
var H:sir;                      sir H; int i, j, n;
  i,j,n:byte;

```

Prezentăm, în continuare, subprogramul *BuildH*, împreună cu o modalitate de apel.

```

4 ...
5 procedure BuildH(k:byte);
6 var i:byte;
7 begin
8   for i:=1 to k-1 do
9     HeapUp(i)
10  end;
11 begin
12   read(n);
13   for i:=0 to n-1 do read(H[i]);
14   BuildH(n);
15   for i:=0 to n-1 do write(H[i])
16 end.

```

```

void HeapUp(int k)
{
  int t;
  if (k <= 0) return;
  t = (k-1)/2;
  if (H[k]>H[t])
  {
    swap(k,t);
    HeapUp(t);
  }
}

```

9. Realizați un subprogram care permite restaurarea unui *maxheap* a cărui rădăcină nu respectă condiția de maxim în raport cu fiile săi. Se presupune că ambi subarborei ai rădăcinii sunt *maxheap*-uri. Subprogramul primește prin intermediul parametrului  $r$ , indicele rădăcinii iar, prin parametrul  $k$ , numărul de elemente din *maxheap*.

*Exemplu:* Pentru  $H=(5,23,44,18,16,6,10)$ , se va afișa  $H=(44,23,10,18,16,6,5)$ .

#### Soluție:

Algoritmul de restaurare a unui *maxheap*, în condițiile date, trebuie să permită coborârea cheii din rădăcină către locația corectă. Această operație se va face prin interschimbarea acestei valori, cu cea mai mare cheie dintre cele două reținute de fiili săi. Procesul continuă până când valoarea coborâtă din rădăcină este mai mare decât ambele chei reținute de fiil vârfului în care s-a ajuns.

Implementarea subprogramului *HeapDw*, care restaurează un *maxheap* conform strategiei prezentate, ia în considerare următoarele declarații:

```

type sir=array[0..100]of byte;    typedef int sir[101];
var H:sir;                      sir H; int i, j, n;
  i,j,n:byte;

```

Subprogramul *swap* realizează interschimbarea a două elemente din *maxheap-ul*  $H$ , situate pe indicații transmise prin parametrii  $k$  și  $t$ .

```

1 ...
2 procedure swap(k,t:byte);
3 var x:byte;
4 begin
5   x:=h[t]; h[t]:=h[k]; h[k]:=x;
6 end;
7
8 procedure HeapDw(r,k:byte);
9 var St,Dr,i:byte;
10 begin
11   if 2*r+1 <= k then begin
12     St := H[2*r+1];
13     if 2*r+2 <= k then
14       Dr := H[2*r+2]
15     else Dr := St-1;
16     if St > Dr then begin
17       i := 2*r+1 end
18     else begin
19       i := 2*r+2 end;
20     if H[r] < H[i] then begin
21       swap(r,i);
22       HeapDw(i,k)
23     end;
24   end; end;
25
26 begin
27   read(n);
28   for i=0;i<n;i++)
29     scanf("%d", &H[i]);
30   BuildH(n);
31   for i=0;i<n;i++)
32     printf("%d ", H[i]);
33 end.

```

```

typedef int sir[101];
sir H; int i, j, n;

```

```

void swap(int k, int t) {
  int x;
  x=H[t]; H[t]=H[k]; H[k]=x;
}

void HeapDw(int r, int k)
{
  int St,Dr,i;
  if (2*r+1<=k) {
    St=H[2*r+1];
    if (2*r+2 <=k) Dr=H[2*r+2];
    else Dr=St-1;
    if (St>Dr) i=2*r+1;
    else i=2*r+2;
    if (H[r]<H[i]) {
      swap(r,i);
      HeapDw(i,k);
    }
  }
}

void main()
{
  scanf("%d", &n);
  for (i=0;i<n;i++)
    scanf("%d", &H[i]);
  BuildH(n);
  for i=0 to n-1 do read(h[i]);
  HeapDw(0,n-1);
  for (i=0;i<n;i++)
    printf("%d ", H[i]);
}

```

**10. Realizați un subprogram care implementează sortarea cu heap-uri (*HeapSort*).**

**Soluție:**

În cazul utilizării *maxheap*-urilor, sortarea se face în ordinea crescătoare a cheilor. Dacă se dorește sortarea descrescătoare a unui sir de valori, se pot folosi *minheap*-urile.

Algoritmul *HeapSort* se bazează pe proprietatea *heap*-urilor, aceea că primul element din vector este cel de valoare maximă. Pe tot parcursul algoritmului, vectorul este împărțit în două părți:

- prima parte conține elementele nesortate, asamblate într-un maxheap;
- ultima parte a vectorului conține restul elementelor afiate în ordine crescătoare. Inițial, aceasta nu conține nici un element.

La fiecare iteratie, cheia rădăcinii (primul element din vector) se interschimbă cu ultimul element din heap.

Astfel, cheia maximă trece în partea a doua a vectorului, acolo unde elementele se află în ordine crescătoare.

Cheia adusă pe prima poziție în vector "strică" proprietatea *maxheap*-ului. Pentru restaurare, se va folosi strategia prezentată în cazul subprogramului *HeapDw*. Procesul se repetă până când heap-ul devine vid.

Pentru o mai bună înțelegere să privim exemplul următor:

Considerăm vectorul  $H = \{32, 78, 8, 56, 23, 45\}$ .

Elementele puse în <i>heap</i>	78	32	56	8	23	45
Pasul 1	56	32	45	8	23	78
Pasul 2	45	32	23	8	56	78
Pasul 3	32	8	23	45	56	78
Pasul 4	23	8	32	45	56	78
Pasul 5	8	23	32	45	56	78
Pasul 6	8	23	32	45	56	78

Implementarea sortării cu *heap*-uri ia în considerare următoarele declarații:

```
type sir=array[0..100]of byte;      typedef int sir[101];
var h:sir;                         sir H; int i, j, n;
i,k,x,j,n:byte;
```

Subprogramul *HeapSort* primește, la apel, prin intermediul parametrului  $k$  ultimul indice din vectorul ce se dorește sortat.

Subprogramul *swap* realizează interschimbarea elementelor din vector situate pe pozițiile 0 și  $k$ .

```
1 .....  
2 procedure HeapSort (k:byte);  
3 begin  
4   while k>0 do begin  
5     swap(0,k);  
6     dec(k);  
7     HeapDw(0,k);  
8   end;  
9 end;  
10  
11 begin  
12   read(n);  
13   for i:= 0 to n-1 do read(h[i]);  
14   BuildH(n);  
15   HeapSort(n-1);  
16   for i:=0 to n-1 do write(h[i])  
17 end.
```

```
.... HeapSort(int k){  
    while (k>0) {  
        swap(0,k);  
        HeapDw(0,k);  
    }  
}  
  
void main(){  
    scanf("%d", &n);  
    for (i=0;i<n;i++)  
        scanf("%d",&H[i]);  
    BuildH(n);  
    HeapSort(n-1);  
    for (i=0;i<n;i++)  
        printf("%d ", H[i]);  
}
```

#### 1.2.4 Probleme propuse

1. Realizați un subprogram care permite memorarea în *heap* a unui arbore binar, ale căruia chei (asociate vârfurilor) sunt introduse în preordine. Lipsa unui fiu se semnalizează prin introducerea valorii 0. Sirul de valori se va citi de la tastatură. Adresa rădăcinii arborelui creat va fi returnată prin intermediul parametrului  $p$ .
2. Realizați un subprogram care permite afișarea în inordine a tuturor cheilor unui arbore binar, situate pe niveluri pare. Rădăcina arborelui se consideră poziționată pe nivelul 0. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina.
3. Considerăm un arbore binar alocat dinamic în *heap* ce memorează valori reale. Realizați un subprogram care determină adâncimea arborelui. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna, prin parametrul întreg  $nr$ , adâncimea acestuia.
4. Considerăm un arbore binar alocat dinamic în *heap* ce memorează valori întregi. Realizați un subprogram care permite afișarea în postordine a cheilor vârfurilor care au doi fiu. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina.
5. Considerăm un arbore binar alocat dinamic în *heap* ce memorează valori întregi. Realizați un subprogram care determină cheile de valoare maximă și minimă. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna, prin parametrii întregi *max* și *min*, valorile extremă determinate.

6. Considerăm un arbore binar alocat dinamic în heap ce memorează valori reale. Realizați un subprogram care determină suma tuturor cheilor memorate în vârfurile situate pe nivelul  $k$ . Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina, prin parametrul întreg  $k$ , nivelul dorit și va returna, prin parametrul real  $s$ , suma determinată.
7. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați un subprogram care efectuează ștergerea din arbore a tuturor vârfurilor ce nu au fii (frunze). Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina.
8. Considerăm un arbore binar alocat dinamic în heap ce memorează valori reale. Realizați o funcție care verifică dacă arborele este plin ( fiecare nivel i este complet, conține  $2^{i-1}$  vârfuri). Subprogramul va primi prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna 1, în caz afirmativ, și 0, în caz contrar.
9. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați un subprogram care determină numărul de frunze situate pe un nivel  $k$ . Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina, prin parametrul întreg  $k$ , nivelul respectiv și va returna numărul determinat, prin parametrul întreg  $nr$ .
10. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați o funcție care verifică dacă arborele conține perechi de valori consecutive  $(x, x+1)$ , memorate în vârfuri aflate în relația tată-fiu. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna 1, în caz afirmativ, și 0, în caz contrar.
11. Fie un arbore binar alocat dinamic în heap ce memorează valori întregi. Creați un subprogram care realizează operația *find-replace*, adică toate cheile din arbore de valoare  $x$  vor fi regăsite și înlocuite cu valoarea  $y$ . Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina iar, prin intermediul parametrilor întregi  $x, y$ , valorile în cauză.
12. Realizați un subprogram care verifică dacă doi arbori binari sunt identici, adică au structură identică și memorează exact aceleași valori pentru orice pereche de vârfuri aflate în aceeași poziție în arbore. Subprogramul va primi, la apel, adresele celor două rădăcini și va returna, prin intermediul parametrului  $ok$ , valoarea 1, dacă arborii sunt identici, sau 0, în caz contrar.
13. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați un subprogram care verifică dacă arborele admite axă de simetrie verticală, adică cei doi subarbore ai rădăcinii au structura în oglindă. Subprogramul va primi, prin intermediul parametrilor  $ps$  și  $pf$ , adresele celor doi fii ai rădăcinii arborilor și va returna, prin intermediul parametrului  $ok$ , valoarea 1, dacă arborele este simetric, sau 0, în caz contrar.
14. Fie un arbore binar alocat dinamic în heap ce memorează valori întregi de cel mult 3 cifre. Realizați funcția *Par* care determină numărul de chei pare dintr-un arbore. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna numărul determinat. Presupunând că se dorește identificarea subarborelui rădăcinii ce conține mai multe valori pare, rezolvați această cerință prin apeluri la funcția *Par*.
15. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați un subprogram care afișează succesiv cheia primului vârf de pe fiecare nivel al arborelui (fiul stâng al rădăcinii, fiul stâng al fiului stâng al rădăcinii, și.a.m.d.). Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina.
16. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați un subprogram care afișează cheile vârfurilor de pe drumul ce pleacă din rădăcina și coboară alternativ prin fiul stâng, apoi prin cel drept. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina.
17. Considerăm un arbore binar alocat dinamic în heap ce memorează valori întregi. Realizați un subprogram care identifică cheia celui de al  $x$ -lea vârf afișat la traversarea în postordine a arborelui. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina.
18. Considerăm un arbore binar alocat dinamic în heap ce memorează valori reale. Realizați o funcție care verifică dacă este arbore de căutare. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna 1, în caz afirmativ, și 0, în caz contrar.
19. Realizați un program care creează un arbore de căutare  $n$  din valori naturale, generate aleator, care sunt cuburi perfecte. Afișați cea mai mică și cea mai mare cheie din arborele generat.
20. Realizați un subprogram care construiește un vector cu informațiile unui arbore de căutare, plasate în ordine crescătoare. Subprogramul va primi, prin intermediul parametrului  $p$ , adresa la care se află memorată rădăcina și va returna, prin doi parametri  $a$  și  $n$ , vectorul construit și numărul lui de elemente.
21. Considerăm un arbore binar de căutare alocat dinamic în heap ce memorează valori naturale mai mici decât 250. Se dorește inserarea în arbore, pentru fiecare frunză, a unei valori egale cu dublul cheii memorate de ea. Realizați un program care construiește un arbore de căutare pe baza a  $n$  valori citite de la tastatură și inserează, în arbore, valorile cerute. Programul va afișa, în preordine, cheile arborelui obținut.

22. Realizați un subprogram care permite ștergerea vârfului cu cheia minimă din subarborele drept (considerat nevid) al unui arbore de căutare. Subprogramul va primi, prin intermediu parametrului  $p$ , adresa la care se află memorată rădăcina.

23. Realizați un program care permite memorarea, în heap, a unui arbore binar, reprezentat cu ajutorul vectorilor  $St$  și  $Dr$ . De la tastatură se citește numărul  $n$  de vârfuri și cele două vectori. Programul va afișa, în preordine, vârfurile arborelui creat.

24. Realizați un program care permite memorarea, în heap, a unui arbore binar, reprezentat cu ajutorul formei ce utilizează paranteze. Lipsa unui fiu este semnalizată cu ajutorul caracterului '\*\*'. De la tastatură se citește numărul  $n$  și sirul de caractere reprezentând forma parantezată a arborelui. Cheile arborelui sunt reprezentate din literele mici ale alfabetului englez. Programul va afișa, în postordine, vârfurile arborelui creat.

25. Considerând cunoscută ordinea de traversare a vârfurilor unui arbore binar, în preordine și în anordine, realizați un program care construiește, în heap, arborele respectiv. Numărul  $n$  de vârfuri și cele două siruri reprezentând parcourgerile în preordine și în anordine se vor citi de la tastatură. Programul va afișa, în postordine, vârfurile arborelui creat.

## 1.3. Structuri de date avansate

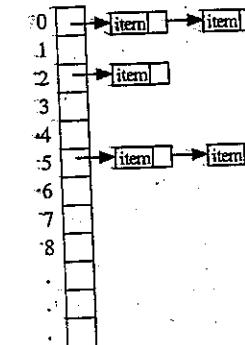
### 1.3.1 Tabele de dispersie (hash)

Multe aplicații necesită o mulțime dinamică pentru care să se aplică numai operațiile specifice pentru dicționare: *insereză*, *căută*, *șterge*. O tabelă de dispersie este o structură eficientă de date pentru implementarea dicționarelor.

O tabelă de dispersie este o generalizare a noțiunii mai simple de tablou. Adresarea directă într-un tablou folosește abilitatea noastră de a examina o poziție arbitrară în timp  $O(1)$ . Adresarea directă este aplicabilă numai în cazul în care ne permitem să alocăm un tablou care are câte o poziție pentru fiecare cheie posibilă. Când numărul cheilor memorate efectiv este relativ mic față de numărul total de chei posibile, tabelele de dispersie devin o alternativă eficientă la adresarea directă într-un tablou, deoarece se folosește un tablou de mărime proporțională cu numărul de chei memorate efectiv. În loc să se folosească direct cheia ca indice în tablou, indicele este calculat pe baza cheii, folosind o funcție de dispersie (hash), care face maparea spațiului de chei într-un spațiu de adrese. Funcția generează o adresă printr-un calcul simplu, aritmetic sau logic asupra cheii sau asupra unei părți a cheii. Datorită faptului că spațiul de chei este de obicei mult mai mare decât spațiul de adrese, se poate întâmpla ca mai multe chei să aibă aceeași adresă. Aceasta se numește *coliziune* între înregistrări.

Pentru rezolvarea coliziunilor, metoda cea mai des folosită este utilizarea listelor simplu înlanțuite pentru fiecare element din spațiu de adresare. Așadar, într-o tabelă de dispersie  $T$ , pe poziția  $i$  se va reține un pointer către începutul listei înlanțuite formate din toate valorile  $x$  pentru care  $h(x)=i$  (unde  $h(x)$  este funcția de dispersie aleasă). Merită menționat că, în cazul cel mai defavorabil, o tabelă de dispersie se poate comporta ca o listă simplu înlanțuită (deci operațiile specificate se execută în timp liniar). Dar în practică, utilizând tabele de dispersie, cele trei operații se execută foarte eficiente.

Operația de inserare se execută în  $O(1)$  (se inserează valoarea  $x$  la începutul listei  $T[h(x)]$ ). Operația de căutare presupune căutarea elementului cu cheia  $x$  în lista înlanțuită  $T[h(x)]$ . Operația de ștergere presupune căutarea elementului cu cheia  $x$  în lista înlanțuită  $T[h(x)]$ , apoi ștergerea din listă. Ultimele două operații au timpul de execuțare proporțional cu lungimea listei  $T[h(x)]$ . Analizând complexitatea în medie, observăm că lungimea listei depinde de căt de uniform distribuite sunt cheile.



Vom prezenta, în continuare, exemple de funcții de dispersie care se comportă bine în practică. Majoritatea funcțiilor de dispersie presupun apartenența cheilor la mulțimea numerelor naturale. Dacă cheile nu sunt numere naturale, se interpretează ca un număr natural (spre exemplu, un sir de caractere poate fi interpretat ca un întreg într-o bază de numerație aleasă convenabil).

#### Exemple de funcții de dispersie:

##### 1. Metoda diviziunii

$$h(k) = k \bmod m$$

În acest caz este indicat să evităm ca  $m$  să fie aproape de o putere a lui 2. Un număr prim apropiat de o putere a lui 2 este adesea o bună alegere pentru  $m$ .

##### 2. Metoda înmulțirii

Se înmulțește cheia  $k$  cu un număr subunitar  $0 < C < 1$ , se consideră partea fraționară a lui  $k * C$ , se înmulțește cu  $m$  și se reține doar partea întreagă a rezultatului.

$$h(k) = [m * \text{frac}(k * C)]$$

Un avantaj al acestei metode este că nu există valori "rele" pentru  $m$ , prin urmare, de obicei  $m$  poate fi o putere a lui 2, pentru a implementa eficient funcția hash. O constantă care se comportă bine în practică este:  $C = (\sqrt{5} - 1)/2 \approx 0.6180339887$ .

##### 3. Metoda probabilistică

Ideea de bază este selectarea funcției de dispersie la întâmplare dintr-o listă predefinită de funcții, la începutul executării programului. Un exemplu care se comportă foarte bine în practică este următorul: se generează aleator un număr



Exemplu:

segment.in

5

2 9 13 9

4 6 12 6

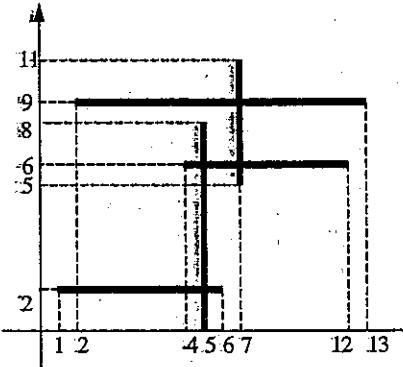
12 6 2

5 0 5 8

7 5 7 11

segment.out

4



Solutie:

Folosind cunoștințe generale de geometrie analitică se poate obține un algoritm  $O(N^2)$ , dar acesta nu se va încadra în limita de timp.

Pentru rezolvarea acestei probleme vom folosi o tehnică cunoscută sub numele de „baleiere” (sweeping) care este comună multor algoritmi de geometrie computațională. În cadrul acestui proces, o dreaptă de baleiere verticală, imaginată traversează multimea obiectelor geometrice, de obicei, de la stânga la dreapta. Baleiera oferă o metodă pentru ordonarea obiectelor geometrice, de obicei, plasându-le într-o structură de date, pentru obținerea relațiilor dintre ele.

Algoritmii de baleiere, în general, gestionează două multimi de date:

1. Starea liniei de baleiere, care dă relația dintre obiectele intersectate de linia de baleiere.
2. Lista punct-eveniment este o secvență de coordonate  $x$ , ordonate de la stânga la dreapta, de obicei, care definesc pozițiile de oprire ale dreptei de baleiere; fiecare astfel de poziție de oprire se numește *punct eveniment*; numai în punctele eveniment se întâlnesc modificări ale stării liniei de baleiere.

Pentru unii algoritmi, lista punct-eveniment este determinată dinamic în timpul executării algoritmului.

În cazul problemei enunțate, vom deplasa o dreaptă de baleiere verticală, imaginată de la stânga la dreapta. Lista punct-eveniment va conține capetele segmentelor orizontale și tipul lor (capăt stânga sau capăt dreapta) și segmentele verticale. Pe măsură ce ne deplasăm de la stânga la dreapta, când întâlnim un capăt stâng, îl inserăm în stările dreptei de baleiere, iar când întâlnim un capăt drept, vom șterge capătul din stările dreptei de baleiere. Când întâlnim un segment vertical, numărul de intersecții ale acestuia cu alte segmente orizontale va fi dat de numărul capetelor de intervale care se află în stările dreptei de baleiere, cuprinse între coordonatele  $y$  ale segmentului vertical.

Astfel, stările dreptei de baleiere sunt o structură de date pentru care avem nevoie de următoarele operații:

- INSEREAZA( $y$ ) : inserează capătul  $y$ ;
- STERGE( $y$ ) : șterge capătul  $y$ ;
- INTEROGARE( $y_1, y_2$ ) : întoarce numărul de capete din intervalul  $[y_1, y_2]$ .

Fie  $MAXC$  valoarea maximă a coordonatelor capetelor de segmente. Folosind un vector pentru a implementa aceste operații descrise mai sus vom obține o complexitate  $O(1)$  pentru primele două operații și  $O(MAXC)$  pentru cea de-a treia. Astfel, complexitatea va fi  $O(N*MAXC)$  în cazul cel mai defavorabil. Putem comprima spațiul  $[0 \dots MAXC]$  observând că doar maximum  $N$  dintre valorile din acest interval conțină, și anume capetele segmentelor orizontale, astfel reducând a treia operație la  $O(N)$ , dar algoritmul va avea complexitatea  $O(N^2)$ , ceea ce nu aduce nici o îmbunătățire față de algoritmul trivial.

Această situație ne îndeamnă să căutăm o structură de date mai eficientă. În continuare vom prezenta o structură de date care oferă o complexitate logaritmică pentru operațiile descrise mai sus.

### ARBORI DE INTERVALE

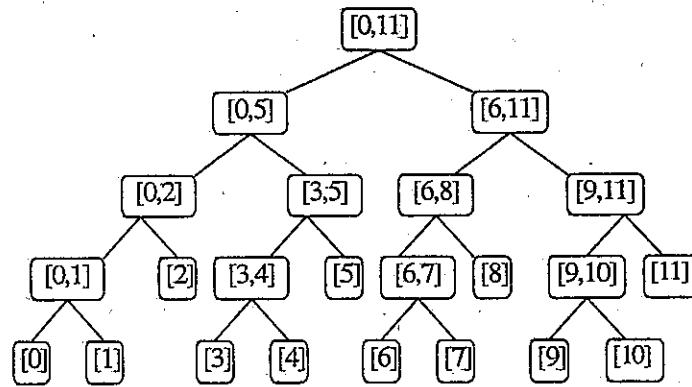
Un *arbore de intervale* este un arbore binar în care fiecare nod poate avea asociată o structură auxiliară (anumite informații). Dându-se două numere întregi  $st$  și  $dr$ , cu  $st < dr$ , atunci arborele de intervale  $T(st, dr)$  se construiește recursiv astfel:

- considerăm rădăcina nod având asociat intervalul  $[st, dr]$ ;
- dacă  $st < dr$ , atunci vom avea asociat subarborele stâng  $T(st, mij)$ , respectiv subarborele drept  $T(mij+1, dr)$ , unde  $mij$  este mijlocul intervalului  $[st, dr]$ .

Intervalul  $[st, dr]$  asociat unui nod se numește *interval standard*. Frunzele arborelui sunt considerate *intervale elementare*, ele având lungimea 1.

Proprietate:

Un arbore de intervale este un arbore binar echilibrat (diferența absolută între adâncimea subarborelui stâng și al subarborelui drept este cel mult 1). Astfel, adâncimea unui arbore de intervale care conține  $N$  intervale este  $\lceil \log_2 N \rceil + 1$ .



#### Operații efectuate asupra unui arbore de intervale

- Actualizarea unui interval într-un arbore de intervale

Vom prezenta pseudocodul unui subprogram recursiv care inserează un interval  $[a,b]$  într-un arbore de intervale  $T(st, dr)$  cu rădăcina în vârful *nod*. Cea mai eficientă metodă de stocare în memorie a unui arbore de intervale este sub forma unui vector folosind aceeași codificare a nodurilor ca la *heap*-uri.

```

1 ACTUALIZARE(nod, st, dr, a, b) // complexitate O(log2 N)
2   daca (a ≤ st) și (dr ≤ b) atunci
3     modifică structura auxiliară din nod
4   altfel
5     mij = (st+dr)/2
6     daca (a <= mij) atunci ACTUALIZARE(2*nod, st, mij, a, b)
7     daca (b > mij) atunci ACTUALIZARE(2*nod+1, mij+1, dr, a, b)
8   actualizează structura auxiliară din nodul nod
  
```

- Interrogarea unui interval într-un arbore de intervale

Vom prezenta pseudocodul unui subprogram recursiv care returnează informațiile asociate unui interval  $[a, b]$  într-un arbore de intervale  $T(st, dr)$  cu rădăcina în nodul *nod*.

```

1 INTEROGARE(nod, st, dr, a, b) // complexitate O(log2 N)
2   daca (a ≤ st) și (dr ≤ b) atunci
3     returnează structura auxiliară din nod
4   altfel
5     mij = (st+dr)/2
6     daca (a <= mij) atunci INTEROGARE(2*nod, st, mij, a, b)
7     daca (b > mij) atunci INTEROGARE(2*nod+1, mij+1, dr, a, b)
8   returnează structura auxiliară din fiul stâng și din fiul drept
  
```

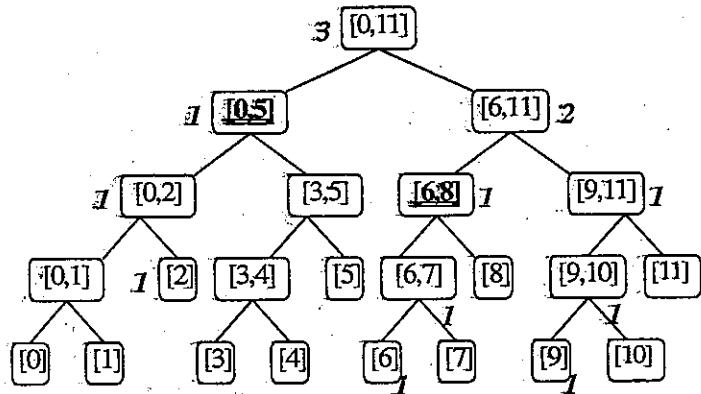
Vom demonstra, în continuare, că operațiile prezentate mai sus au complexitatea  $O(\log_2 N)$  pentru un arbore de  $N$  intervale. Este posibil, ca într-un nod, să aibă loc apel atât în fiul stâng, cât și în cel drept. Acest lucru produce un cost adițional doar prima dată când are loc. După prima „răpare în două”, oricare astfel de operație nu va aduce cost adițional, deoarece unul dintre fiii va fi mereu inclus complet în intervalul  $[a,b]$ . Cum înălțimea arborelui pentru  $N$  intervale este egală cu  $\lceil \log_2 N \rceil + 1$ , complexitatea operațiilor va fi tot  $O(\log_2 N)$ .

Pentru a reține în memorie un arbore de intervale pentru  $N$  valori, vom avea nevoie de  $N+N/2+N/4+N/8\dots=2^kN-1$  locații de memorie (sunt  $2^kN-1$  noduri). Deoarece arborele nu este complet, trebuie verificat de fiecare dată dacă fiile unui nod există în arbore (această verificare a fost omisă în pseudocodul de mai sus), altfel s-ar încerca accesarea de valori din vector care nu există. Dacă memoria disponibilă în timpul concursului este suficientă, se poate declara vectorul care reține arborele de intervale de lungime  $2^k$  astfel încât  $2^k \geq 2^kN-1$ , simulând astfel un arbore complet și nefiind necesare verificările menționate mai sus.

#### Soluția problemei (continuare)

Vom folosi un arbore de intervale pentru a simula operațiile, pe care le făceam înainte pe un vector obișnuit, în timp logarithmic. Astfel, în fiecare nod din arborele din intervale vom reține căte capete există în acel interval. Primele două operații vor fi implementate folosind procedura *ACTUALIZARE()* de mai sus, pentru intervalul  $[y, y]$  în arborele  $T(0, MAXC)$  și adunând +1, respectiv -1 la fiecare nod actualizat. Cea de-a treia operație poate fi realizată folosind procedura *INTEROGARE()* pe intervalul  $[y_1, y_2]$ . Astfel, complexitatea se reduce la  $O(N * \log_2 MAXC)$ . Folosind aceeași tehnică de „comprimare” a coordonatelor se poate obține o complexitate  $O(N * \log_2 N)$ .

În figura următoare este descrisă structura arborelui de intervale, după actualizarea pentru intervalele  $[2,2]$ ,  $[6,6]$  și  $[9,9]$ . Sunt marcate intervalele care conduc la obținerea numărului de segmente intersectate de primul segment vertical, obținute în urma interogării pe intervalul  $[0,8]$ .



Implementarea în limbaj (FreePascal, GCC) a algoritmului este prezentată în continuare:

```

1 type entry = record
2   x, y1, y2, sgn:integer;
3 end;
4 sir = array[0..50000]of entry;
5 var a:sir;
6 i, n, x1, y1, x2, y2, v :
word;
7 T : array[0..131072]of word;
8 Res: Int64;
9
10
11 function cmp (i,j: entry):integer;
12 begin
13   if i.x>>j.x then cmp:=i.x-j.x
14   else cmp:=j.sgn - i.sgn;
15 end;
16
17 function query(
18   n,l,r,a,b:word):word;
19 var m, tt:word;
20 begin
21   tt := 0;
22   if (a<=l)and(r<=b)then tt:=T[n]
23   else begin
24     m := (l + r) div 2;
25     if (a <= m) then
26       tt:=tt+query(2*n, l, m, a, b);
27     if (b > m) then
28       tt := tt +
29         query(2*n+1, m+1, r, a, b);
30   end;
31   query := tt;
32 end;

```

```

32 procedure update(n,l,r,p,v: word);
33 var m : word;
34 begin
35   m := (l + r) div 2;
36   inc(T[n],v);
37   if (l <= r) then begin
38     if (p <= m) then
39       update(2*n, l, m, p, v)
40     else
41       update(2*n+1,m+1, r, p, v);
42   end;
43 end;
44
45 procedure QuickSort(var A:
46   ..sir; Lo, Hi: Integer);
47 ...{corp procedure QuickSort}
48
49 begin
50   assign(input, 'segment.in');
51   reset(input);
52   readln(n);
53   v:=0;
54   for i := 0 to N - 1 do begin
55     readln(x1, y1, x2, y2);
56     if (y1 = y2) then begin
57       A[v].x:= x1;
58       A[v].sgn:= +1;
59       A[v].y1:= y1;
60       inc(v);
61       A[v].x:= x2;
62       A[v].sgn:= -1;
63       A[v].y1 := y1;
64       inc(v);
65     end
66     else
67       if (x1 = x2) then begin
68         A[v].x:= x1; A[v].sgn:= 0;
69         A[v].y1:=y1; A[v].y2:= y2;
70         inc(v); end;
71   end;
72   close(input);
73   QuickSort(A, 0, v-1);
74   Res:=0;
75   for i := 0 to v - 1 do begin
76     if (A[i].sgn = 0) then
77       Res:=Res + query(1, 0,49999,
78                         A[i].y1,A[i].y2)
79     else
80       update(1,0,49999,
81             A[i].y1,A[i].sgn)
82   end;
83   assign(output, 'segment.out');
84   rewrite(output); writeln(Res);
85   close(output);
86 end.

```

### 1.3.3 Arbori indexati binar

Arborii indexati binar reprezintă o structură de date care permite efectuarea următoarelor operații în complexitatea  $O(\log N)$  pentru un vector de  $N$  elemente:

- ADAUGĂ( $x, y$ ) – adună la elementul de pe poziția  $x$  din vector, valoarea  $y$ ;
- SUMĂ( $x$ ) – calculează suma primelor  $x$  elemente din vector (se presupune că vectorul este indexat de la 1 la  $N$ ).

Operațiile descrise mai sus pot fi efectuate în complexitatea menționată folosind un arbore de intervale (descriș în capitolul anterior), dar folosirea unui arbore indexat binar oferă anumite avantaje:

- implementarea este mai ușoară;
- se folosesc fix  $N$  locații de memorie pentru stocarea arborelui;
- operațiile se pot extinde ușor pentru a rezolva aceeași problemă în mai multe dimensiuni.

Din păcate, această structură de date nu oferă aceeași flexibilitate precum un arbore de intervale, existând seturi de operații care pot fi efectuate în complexitate logaritmică folosind doar arbori de intervale.

Eficiența acestei structuri de date nu are legătură cu echilibrarea arborelui precum alte structuri de date convenționale, ci din modul de indexare a elementelor, ținând cont de reprezentarea binară (de unde și *numele de arbore indexat binar*). Ideea de bază este că, precum un număr  $n$  poate fi exprimat ca sumă de puteri ale lui 2, așa și un interval  $[1..n]$  poate fi exprimat ca reuniunea unor subintervale de lungimi egale cu puteri ale lui 2.

Vom lua în continuare, ca exemplu, intervalul  $[1..11]$ :

$$11 = 2^3 + 2^1 + 2^0 = 1011_{(2)}$$

Eliminând cel mai puțin semnificativ bit de 1 din reprezentarea lui 11 în baza 2, constatăm că suma pentru intervalul  $[1..11]$  poate fi calculată astfel:

$$[1..11] \quad (11 = 1011_{(2)}) = [1..10] + [11..11]$$

$$[1..10] \quad (10 = 1010_{(2)}) = [1..8] + [9..10]$$

$$[1..8] \quad (8 = 1000_{(2)})$$

Astfel, dacă se asociază fiecărui element  $x$  din arbore un interval  $[x-2^k+1..x]$ , unde  $k$  reprezintă numărul zecourilor terminale din reprezentarea binară a lui  $x$ , fiecare interval de forma  $[1..x]$  poate fi exprimat ca reuniunea a cel mult  $\log_2 N$  intervale, folosind procedeul (descriș mai sus) de eliminare succesivă a celui mai nesemnificativ bit de 1. Având stabilită această structură, operația SUMĂ( $x$ ) se efectuează însumând intervalele din descompunerea lui  $[1..x]$ .

Considerăm un arbore indexat binar cu 15 elemente (distribuția elementelor este reprezentată în figura alăturată). Pentru a actualiza valoarea unui element  $x$  (operația ADAUGĂ( $x, y$ )) se actualizează întâi intervalul asociat lui  $x$  ( $[x-2^k+1..x]$ ) cât și restul intervalelor care îl conțin pe  $x$ . Acest lucru se efectuează adăugând succesiv cel mai nesemnificativ bit de 1. Spre exemplu, pentru a actualiza elementul de pe poziția 9, se modifică următoarele elemente (se consideră că sunt 15 elemente):

$$9 \quad (9 = 1001_{(2)}) - [9..9]$$

$$10 \quad (10 = 1010_{(2)}) - [9..10]$$

$$12 \quad (10 = 1100_{(2)}) - [9..12]$$

Pentru a determina în  $O(1)$  cel mai nesemnificativ bit de 1 al unui număr, se folosește expresia  $(x \text{ and } (x-1)) \text{ xor } x$  (Pascal) /  $(x \& (x-1)) ^ x$  (C/C++). Explicația este simplă:

$$\begin{aligned} x &= ?...?10...0 \\ x-1 &= ?...?01...1 \\ x \& (x-1) &= ?...?00...0 \\ (x \& (x-1))^x &= 0...010...0 \end{aligned}$$

Deși această structură de date nu este la fel de flexibilă ca un arbore de intervale, ea poate fi extinsă să suporte și alte operații:

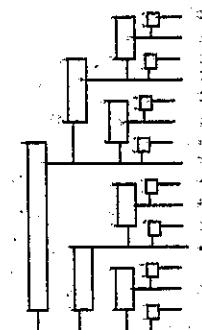
- SUMĂ( $x, y$ ) – suma elementelor cu poziții între  $x$  și  $y$ : se poate scrie ca suma elementelor din intervalul  $[1..x]$  minus suma elementelor din intervalul  $[1..y-1]$ ;
- MODIFICĂ( $x, y$ ) – actualizează valoarea elementului  $x$  la  $y$ , dacă aceasta era mai mică decât  $y$ ;
- MAXIM( $x$ ) – determină maximul din primele  $x$  elemente.

ACESTE operații pot fi implementate înlocuind operația de adunare (" $+$ ") cu cea de maxim.

- ADAUGĂ( $x, y, z$ ) – adaugă la toate elementele cu poziții între  $x$  și  $y$  valoarea  $z$ ;
- AFLĂ( $x$ ) – determină valoarea elementului de pe poziția  $x$ .

Fie  $V$  vectorul asupra căruia se fac operațiile descrise. Se va folosi un arbore indexat binar pentru a menține un alt vector  $V'$  cu proprietatea că  $V'[i] = V[1] + V[2] + \dots + V[i]$ . Pentru a adăuga la toate elementele cu poziții între  $x$  și  $y$  valoarea  $z$ , este de ajuns să se adauge la elementul  $x$  în  $V'$  valoarea  $z$  și la elementul  $y+1$  din  $V'$  valoarea  $-z$ .

Pentru a afla un element  $V[i]$  se face suma primelor  $i$  elemente din  $V'$ .



## Aplicatie - Arbori indexati binar

Se consideră un teren de formă dreptunghiulară având lungimea și lățimea  $N \leq 1000$ . Terenul este împărțit în regiuni de formă pătrată având latura egală cu unitatea. În fiecare moment, într-o anumită regiune pot fi plantați sau tăiați pomi. De asemenea, în fiecare moment proprietarul terenului ar putea cere informații referitoare la numărul total al pomilor care se află într-o regiune dreptunghiulară a terenului, cu unul din colțuri în  $(1, 1)$ . Implementați eficient aceste operații!

### Soluție:

Problema poate fi văzută ca o variantă în 2 dimensiuni a problemei discutată inițial.

Totuși, rezolvarea cu arbori de intervale este destul de dificilă. Vom arăta în continuare cum poate fi folosit un arbore indexat binar pentru a obține o complexitate  $O(\log^2 N)$  pe operație. Se asociază fiecărui element  $[i, j]$  din matrice intervalul bidimensional  $[i-2^p+1..i, j-2^q+1..j]$ , unde  $p, q$  reprezintă numărul zerourilor terminale din reprezentarea binară a lui  $i$ , respectiv  $j$ .

Operațiile se vor implementa identic ca în cazul unidimensional.

```

1 function bit(x:integer):integer;
2 begin
3   bit:=(x and (x-1)) xor x and;
4
5 procedure adauga(x,y,z:integer);
6 var i,j:integer;
7 begin i:=x;
8   while i<=n do begin
9     j:=y;
10    while j<=n do begin
11      inc(a[i,j],z);
12      inc(j,bit(j));
13    end;
14    inc(i,bit(i));
15  end;
16 end;
17
18 function suma(x,y:integer):integer;
19 var i,j,s:integer;
20 begin
21   s:=0; i:=x;
22   while i>0 do begin
23     j:=y;
24     while j>0 do begin
25       inc(s,a[i,j]); dec(j,bit(j));
26     end;
27     dec(i,bit(i));
28   end;
29   suma:=s;
30 end;
31 ...

```

## 1.3.4 Arbori eficienți de căutare – treap-uri

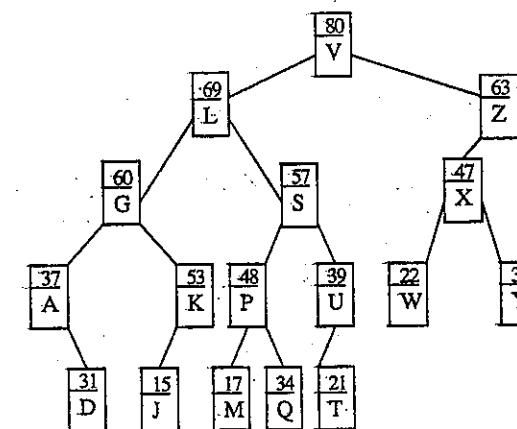
Treap-urile reprezintă o variantă de arbori binari de căutare randomizați. O analiză riguroasă arată că operațiile efectuate pe un treap au complexitatea  $O(\log N)$ , dar cunoștințele matematice necesare pentru această demonstrație depășesc scopul acestei prezentări.

Principalele operații suportate de un treap sunt:

- INSEREAZĂ( $x$ ) – inserează valoarea  $x$  în arbore;
- CAUTĂ( $x$ ) – verifică dacă valoarea  $x$  există în arbore;
- ȘTERGE( $x$ ) – șterge valoarea  $x$  din arbore;
- IMPARTE( $x$ ) – împarte arborele în doi arbori, unul conținând doar valori  $< x$ , iar celălalt doar valori  $> x$ ;
- UNEȘTE( $T_1, T_2, x$ ) – unește două treap-uri într-unul singur (se presupune că unul conține doar valori  $< x$ , iar celălalt, doar valori  $> x$ ).

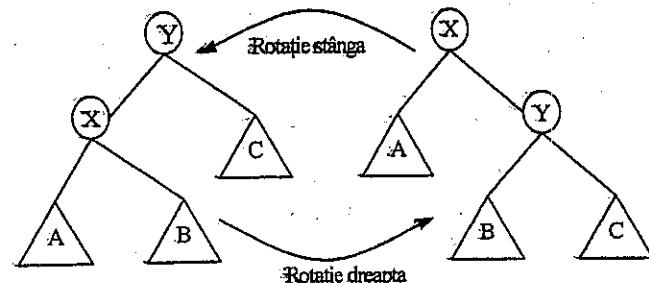
Fără de alte variante de arbori de căutare precum arborii roșu-negru, arborii splay, implementarea treap-urilor necesită mult mai puțin timp.

Pentru fiecare nod din arbore se asociază o valoare ca în orice arbore binar de căutare. În plus, fiecare nod va mai avea o *prioritate*, care este un număr aleator ales independent de celelalte noduri. Nodurile din treap sunt ordonate astfel încât, ținând cont de valori, arborele este un arbore binar de căutare, iar ținând cont de priorități, arborele este un max-heap.



Un treap cu litere ca valori

Operația de bază folosită pentru păstrarea proprietății de arbore de căutare și de heap este *rotatia*. O rotație într-un arbore de căutare (indiferent de direcție – stânga sau dreapta) păstrează validă proprietatea de arbore de căutare.



### *Rotatii într-un arbore binar de căutare*

În continuare, vom descrie cum se efectuează operațiile prezentate mai sus, într-un treapta:

- INSEREAZĂ( $x$ ) – se inserează valoarea  $x$  împreună cu o prioritate aleatorie, în mod normal, ca într-un arbore binar de căutare; pentru a menține proprietatea de max-heap se folosesc rotații cât timp nodul curent are prioritate mai mare decât tatăl său;
  - CAUTĂ( $x$ ) – se efectuează o căutare normală ca într-un arbore binar de căutare;
  - STERGE( $x$ ) – după ce se găsește valoarea  $x$  în arbore, se folosesc rotații pentru a scufunda nodul respectiv (se rotește astfel încât fiul, care are prioritate mai mare, să ajungă în vârf) până când devine o frunză și se elimină;
  - ÎMPARTE( $x$ ) – se inserează, în treap, un nod cu cheie  $x$  și prioritate  $\infty$ ; acest element va deveni rădăcina treap-ului; fiul stâng va conține chei cu valori  $< x$ , iar fiul drept, elemente cu valori  $> x$ ;
  - UNEȘTE( $T_1, T_2, x$ ) – se creează un nod nou cu valoarea  $x$  și prioritatea  $\infty$ , având ca fiu stâng  $T_1$  și ca fiu drept,  $T_2$  și apoi se șterge acest nod.

Implementarea tuturor operațiilor descrise este prezentată în continuare:

```

1 ...  

2 const inf=16191;  

3 type treap=^nod;  

4     nod=record  

5         v,p:integer;  

6         st,dr:treap;  

7     end;  

8 var r,null:treap;  

9

```

```

10 procedure init;
11 begin
12   new(null); x:=null;
13   null^.v:=-inf; null^.p:=-inf;
14   null^.st:=nil; null^.dr:=nil;
15 end;
16
17 procedure rot_st(var n:treap);
18 var t:treap;
19 begin
20   t:=n^.st; n^.st:=t^.dr;
21   t^.dr:=n; n:=t;
22 end;
23
24 procedure rot_dr(var n:treap);
25 var t:treap;
26 begin
27   t:=n^.dr; n^.dr:=t^.st;
28   t^.st:=n; n:=t;
29 end;
30
31 procedure insereaza(var n:treap;
32 v,p:integer);
33 begin
34   if n=null then begin
35     new(n); n^.v:=v; n^.p:=p;
36     n^.st:=null; n^.dr:=null;
37     exit;
38   end;
39   if v<n^.v then begin
40     insereaza(n^.st,v,p);
41     if n^.st^.p=n^.p then rot_st(n);
42   end;
43   if v>n^.v then begin
44     insereaza(n^.dr,v,p);
45     if n^.dr^.p=n^.p then rot_dr(n);
46   end;
47 end;
48
49 procedure sterge(var n:treap;
50 v:integer);
51 begin
52   if n=null then exit;
53   if v<n^.v then sterge(n^.st,v);
54   if v>n^.v then sterge(n^.dr,v);
55   if v=n^.v then begin
56     if n^.st^.p=n^.dr^.p then rot_st(n)
57     else rot_dr(n);
58     if n<>null then sterge(n, v)
59     else begin
60       dispose(n^.st);
61       n^.st:=nil;
62     end;
63   end;
64 end;
65
66 void init()
67 {
68   R=NIL=new treap;
69   NIL->v=NIL->p=-INF;
70   NIL->st=NIL->dr=NULL;
71 }
72
73 void rot_st(treap *&n)
74 {
75   treap *t=n->st;
76   n->st=t->dr; t->dr=n; n=t;
77 }
78
79 void rot_dr(treap *&n)
80 {
81   treap *t=n->dr;
82   n->dr=t->st; t->st=n; n=t;
83 }
84
85 void insereaza(treap *&n,
86                 int v, int p)
87 {
88   if (n == NIL)
89   {n=new treap;
90    n->v=v; n->p=p;
91    n->st=n->dr=NIL;
92    return;
93   }
94   if (v < n->v)
95   {
96     insereaza(n->st, v, p);
97     if(n->st->p>n->p) rot_st(n);
98   }
99   if (v > n->v) {
100     insereaza(n->dr, v, p);
101     if(n->dr->p>n->p) rot_dr(n);
102   }
103 }
104
105 void sterge(treap *&n, int v)
106 {
107   if (n == NIL) return;
108   if (v<n->v) sterge(n->st, v);
109   if (v>n->v) sterge(n->dr, v);
110   if (v == n->v) {
111     if (n->st->p > n->dr->p)
112       rot_st(n);
113     else
114       rot_dr(n);
115     if (n != NIL)
116       sterge(n, v);
117     else {
118       free(n->st);
119       n->st=NULL;
120     }
121   }
122 }

```

```

65 function cauta(n:treap;
66 v:integer):boolean;
67 begin
68 if n=null then cauta:=false
69 else if v<n^.v then
70 cauta:=cauta(n^.st,v)
71 else if v>n^.v then
72 cauta:=cauta(n^.dr,v)
73 else cauta:=true;
74 end;
75
76 procedure imparte(var
77 n,t1,t2:treap; v:integer);
78 begin
79 inseraza(n,v,INF);
80 t1:=n^.st;
81 t2:=n^.dr;
82 end;
83
84 function uneste(var t1,t2:treap;
85 v:integer):treap;
86 var n:treap;
87 begin
88 new(n);
89 n^.v:=v;
90 n^.p:=inf;
91 n^.st:=t1;
92 n^.dr:=t2;
93 sterge(n,v);
94 uneste:=n;
95 end;
96 ...

```

```

int cauta(treap *n, int v)
{
    if (*n == NIL) return 0;
    if (v < n->v)
        return cauta(n->st, v);
    if (v > n->v)
        return cauta(n->dr, v);
    return 1;
}

void imparte(treap *&n, treap
*&t1, treap *&t2, int v){
inseraza(n, v, INF);
t1 = n->st;
t2 = n->dr;
}

treap* uneste(treap *t1, treap
*t2, int v) {
    treap *n;
    n = new treap;
    n->v=v;
    n->p=INF;
    n->st=t1;
    n->dr=t2;
    sterge(n, v);
    return n;
}
...

```

### 1.3.5 Probleme propuse

1 (\*\*). Să se determine toate aparițiile unei matrice  $P$  de dimensiuni  $m \times m$  ca submatrice într-o matrice  $T$  de dimensiuni  $n \times n$ .

2 (\*\*). Pe o foaie de matematică sunt desenate  $4 \leq N \leq 1000$  puncte. Câte trapeze se pot forma cu vârfurile în aceste puncte? Un trapez este un patruăter convex cu cel puțin două laturi paralele.

Pe prima linie din fișierul de intrare *trapez.in* se găsește numărul natural  $N$ . Pe următoarele  $N$  linii se găsesc perechi de numere naturale reprezentând coordonatele punctelor.

Pe prima linie din fișierul de ieșire *trapez.out* se va găsi numărul de trapeze care se pot forma. Coordonatele punctelor sunt numere întregi din intervalul  $[0, 2.000.000.000]$ . Oricare trei puncte sunt necoliniare.

*Exemplu:*

*trapez.in*  
5  
0.0  
0.1  
1.4  
2.0  
3.1

*trapez.out*  
1

(<http://infoarena.devnet.ro>)

3 (\*\*). Pe o foaie de matematică sunt desenate  $3 \leq N \leq 1500$  puncte. Câte triunghiuri echilaterale se pot forma cu vârfurile în aceste puncte?

Pe prima linie din fișierul de intrare *triang.in* se găsește numărul natural  $N$ . Pe următoarele  $N$  linii se găsesc perechi de numere reale reprezentând coordonatele punctelor.

Pe prima linie din fișierul de ieșire *triang.out* se va găsi numărul de triunghiuri echilaterale care se pot forma.

Coordinatele punctelor sunt numere reale din intervalul  $[-10.000, 10.000]$ .

*Exemplu:*

*triang.in*  
3  
0.0  
4.0  
2.3 4.6 4.1 0.1 6

*triang.out*  
1

(<http://infoarena.devnet.ro>)

4 (\*\*). Se consideră  $3 \leq N \leq 1000$  puncte în plan. Să se determine mulțimea de cardinal maxim cu proprietatea că toate punctele pe care aceasta le conține sunt coliniare.

Numărul  $N$ , reprezentând numărul de puncte, se află pe prima linie a fișierului *colin.in*. Pe următoarele  $N$  linii se află câte două numere întregi reprezentând coordonatele punctelor, valori din intervalul  $[-10000, 10000]$ .

Pe prima linie a fișierului *colin.out* se va afișa cardinalul mulțimii obținute.

*Exemplu:*

*colin.in*  
7  
10 20  
0 0  
44 90  
20 30  
22 45  
11 20  
66 135

*colin.out*  
4

5 (\*\*\*) Se consideră  $N \leq 50.000$  dreptunghiuri în plan, fiecare având laturile paralele cu axele OX/OY. Lungimea marginilor reuniunii (conturul) tuturor dreptunghiurilor se va numi perimetru. Să se calculeze perimetruul celor  $N$  dreptunghiuri.

În fișierul „*drept.in*” se găsește pe prima linie numărul  $N$  de dreptunghiuri, iar pe fiecare din următoarele  $N$  linii câte patru numere naturale mai mici ca 50.000, reprezentând coordonatele carteziene ale colțului stânga sus, respectiv dreapta jos ale fiecărui dreptunghi.

Rezultatul se va scrie în fișierul „*drept.out*”.

*Exemplu:*

<i>drept.in</i>	<i>drept.out</i>
3	44
3 8 8 3	
6 10 12 6	
12 4 15 1	

6 (\*\*\*) Se dău  $N \leq 100.000$  puncte în plan de coordonate numere naturale mai mici ca 2.000.000.000. Să se răspundă la  $M \leq 1.000.000$  întrebări de forma „câte puncte dintre cele  $N$  există în dreptunghiul cu colțul stânga sus în  $(x_1, y_1)$  și colțul dreapta jos în  $(x_2, y_2)$ ?”

În fișierul „*puncte.in*” se vor găsi pe prima linie numerele  $N$  și  $M$ . Pe următoarele  $N$  linii se găsesc coordonatele punctelor. Pe următoarele  $M$  linii se vor găsi câte patru numere naturale reprezentând coordonatele colțurilor dreptunghiurilor.

În fișierul „*puncte.out*” se vor găsi  $M$  numere naturale reprezentând răspunsurile la întrebări.

*Exemplu:*

<i>puncte.in</i>	<i>puncte.out</i>
6 1	2
2 8	
5 3	
6 5	
8 7	
8 1	
10 10	
4 8 9 4	

7 (\*\*\*) Se consideră  $N \leq 50.000$  dreptunghiuri în plan, fiecare având laturile paralele cu axele OX/OY. Să se calculeze aria ocupată de reuniunea celor  $N$  dreptunghiuri.

În fișierul „*drept2.in*” se găsește, pe prima linie, numărul  $N$  de dreptunghiuri, iar pe fiecare dintre următoarele  $N$  linii, câte patru numere naturale mai mici ca 50.000, reprezentând coordonatele carteziene ale colțului stânga sus, respectiv dreapta jos, ale fiecărui dreptunghi. Rezultatul se va scrie în fișierul „*drept2.out*”.

8 (\*\*\*\*). Se consideră  $N \leq 50.000$  puncte în plan de coordonate numere naturale mai mici ca 50.000. Să se determine unde se poate așeza un dreptunghi de lungime  $DX$  și lățime  $DY$  astfel încât numărul de puncte incluse în dreptunghi să fie maxim.

În fișierul „*puncte.in*” se găsește pe prima linie numerele  $N$ ,  $DX$  și  $DY$ . Pe următoarele  $N$  linii se găsesc coordonatele punctelor.

În fișierul „*puncte.out*” se vor găsi cinci numere naturale reprezentând coordonatele colțurilor stânga sus și dreapta jos ale așezării dreptunghiului și numărul maxim de puncte din dreptunghi.

9 (\*\*\*\*). Se consideră  $N \leq 100.000$  puncte în plan de coordonate numere naturale mai mici ca 100.000. Să se determine unde se pot așeza două dreptunghiuri de lungime  $DX$  și lățime  $DY$ , fără să se intersecteze, astfel încât numărul de puncte incluse în cele două dreptunghiuri să fie maxim.

În fișierul „*puncte2.in*” se găsește pe prima linie numerele  $N$ ,  $DX$  și  $DY$ . Pe următoarele  $N$  linii se găsesc coordonatele punctelor.

În fișierul „*puncte2.out*” se vor găsi nouă numere naturale reprezentând coordonatele colțurilor stânga sus și dreapta jos ale așezării primului dreptunghi, respectiv ale celui de-al doilea, cât și numărul maxim de puncte incluse în ambele dreptunghiuri.

10 (\*\*\*\*). Se consideră  $N \leq 250.000$  dreptunghiuri în plan fiecare având laturile paralele cu axele OX/OY, care nu se intersectează și nu se ating, dar care pot fi incluse unul în altul. Se numește „închisoare” un dreptunghi înconjurat de alte dreptunghiuri. Să se determine numărul maxim de dreptunghiuri de care poate fi înconjurate o „închisoare” și câte astfel de „închisori” maxime există.

În fișierul „*inchis.in*” se găsește pe prima linie numărul  $N$  de dreptunghiuri, iar pe fiecare dintre următoarele  $N$  linii, câte patru numere naturale mai mici ca 1.000.000.000, reprezentând coordonatele carteziene ale colțului stânga sus, respectiv dreapta jos ale fiecărui dreptunghi.

În fișierul „*inchis.out*” se găsesc două numere naturale: numărul maxim de dreptunghiuri de care poate fi înconjurate o „închisoare” și câte astfel de „închisori” maxime există.

11 (\*\*\*) Se dău  $N \leq 3500$  cutii paralelipipedice prin dimensiunile lor ( $X$ ,  $Y$  și  $Z$ ). Se știe că o cutie se poate pune în alta doar dacă toate dimensiunile ei sunt strict mai mici decât cele ale cutiei în care va fi băgată. Se cere numărul *maxim* de cutii ce pot fi selectate din cele  $N$ , astfel încât ele să poată fi „cuibărite” (o cutie va conține o cutie, care la rândul ei va conține o alta, și.a.m.d. până la cea mai mică care nu va mai conține nimic).

Prima linie a fișierului *cutii.in* va conține  $N$  și  $T$ , reprezentând numărul de cutii, respectiv numărul de teste care vor urma. Pentru fiecare din cele  $T$  teste vor urma câte  $N$  linii, conținând trei numere reprezentând dimensiunile fiecărei cutii.

Fișierul *cutii.out* va conține  $T$  linii, pe fiecare linie un număr reprezentând numărul maxim de cutii ce pot fi alese pentru fiecare test.

*Exemplu:*

<i>cutii.in</i>	<i>cutii.out</i>
3 2	3
1 1 1	2
2 2 2	
3 3 3	
1 2 2	
2 1 1	
3 3 3	

(<http://infoarena.devnet.ro>)

**12 (\*\*\*\*).** Lui Algorel îi plac mult sirurile de numere naturale cu proprietăți căt mai ciudate. Căutând astfel de ciudătenii ale informaticii, a găsit printr-o carte prăfuită de vreme un nou tip de sir denumit evantai. Un evantai este un sir cu un număr par de termeni,  $E_1, E_2, \dots, E_{2K}$  cu următoarea proprietate:

$$E_1 + E_{2K} > E_2 + E_{2K-1} > \dots > E_K + E_{K+1}.$$

Fiind dat un sir de numere naturale distincte  $A_1, A_2, \dots, A_N$ , Algorel vrea să afle căte subșiruri ale acestuia sunt evantaie.

Prima linie a fișierului *evantai.in* conține numărul întreg  $N \leq 700$ , reprezentând numărul de elemente ale sirului. Următoarele  $N$  linii conțin, în ordine, elementele sirului  $A$ . Elementele sirului sunt numere întregi distincte cuprinse între 1 și 1000. Pe prima linie a fișierului *evantai.out* se va afla un singur număr întreg  $C$ , reprezentând numărul de subșiruri evantaie. Rezultatul va fi afișat modulo 30103.

*Exemplu:*

<i>evantai.in</i>	<i>evantai.out</i>
4	
1	
2	
3	
6	7

(<http://infoarena.devnet.ro>)

**13 (\*\*\*\*\*).** Dându-se un graf neorientat cu  $N \leq 50.000$  noduri și  $M \leq 100.000$  muchii, să se implementeze eficient următoarele operații (vor fi maximum 200.000 operații):

- ADAUGĂ( $x, y$ ) – se adaugă o muchie în graf între nodurile  $x$  și  $y$ ;
- STERGE( $x, y$ ) – se șterge din graf muchia dintre nodurile  $x$  și  $y$ ;
- COMPONENTĂ( $x, y$ ) – se afișează *true* dacă nodurile  $x$  și  $y$  sunt în aceeași componentă conexă, sau *false* în caz contrar.

**14 (\*\*\*\*\*).** Se consideră un vector de  $N \leq 50.000$  elemente numere întregi. Să se implementeze eficient următoarele operații (vor fi maximum 10.000 operații):

- SCHIMBĂ( $x, y$ ) – setează valoarea elementului de pe poziția  $x$  la  $y$ ;
- SELECTEAZĂ( $x, y, z$ ) – afișează al  $z$ -lea număr din vector, considerând doar elementele cu poziții între  $x$  și  $y$ .

## CAPITOLUL 2

### Teoria grafurilor

#### 2. 1. Noțiuni introductive

##### 2.1.1 Terminologie

*Graf*  $\Leftrightarrow$  orice mulțime finită  $V$ , prevăzută cu o relație binară internă  $E$ . Notăm graful cu  $G=(V,E)$ .

*Graf neorientat*  $\Leftrightarrow$  un graf  $G=(V,E)$ , în care relația binară este simetrică: dacă  $(v,w) \in E$ , atunci  $(w,v) \in E$ .

*Graf orientat*  $\Leftrightarrow$  un graf  $G=(V,E)$ , în care relația binară nu este simetrică.

*Nod*  $\Leftrightarrow$  element al mulțimii  $V$ , unde  $G=(V,E)$  este un graf neorientat.

*Vârf*  $\Leftrightarrow$  element al mulțimii  $V$ , unde  $G=(V,E)$  este un graf orientat sau neorientat.

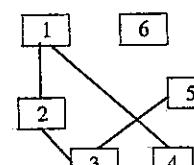
*Muchie*  $\Leftrightarrow$  element al mulțimii  $E$  ce descrie o relație existentă între două vârfuri din  $V$ , unde  $G=(V,E)$  este un graf neorientat;

*Arc*  $\Leftrightarrow$  element al mulțimii  $E$  ce descrie o relație existentă între două vârfuri din  $V$ , unde  $G=(V,E)$  este un graf orientat;

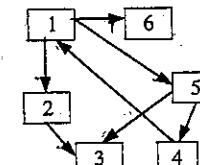
Arcele sunt parcurse în direcția dată de relația vârf  $\rightarrow$  succesor\_direct.

Muchiile unui graf neorientat sunt considerate ca neavând direcție, deci pot fi parcuse în ambele sensuri.

*Adiacență*  $\Leftrightarrow$  Vârful  $w$  este adiacent cu  $v$  dacă perechea  $(v,w) \in E$ . Într-un graf neorientat, existența muchiei  $(v,w)$  presupune că  $w$  este adiacent cu  $v$  și  $v$  adiacent cu  $w$ .



*Graf neorientat*



*Graf orientat*

În exemplele din figura de mai sus, vârful 1 este adiacent cu 4, dar 1 și 3 nu reprezintă o pereche de vârfuri adiacente.

*Incidență*  $\Leftrightarrow$  o muchie este incidentă cu un nod dacă îl are pe acesta ca extremitate. Muchia  $(v,w)$  este incidentă în nodul  $v$ , respectiv  $w$ .

*Incidență spre interior*  $\Leftrightarrow$  Un arc este incident spre interior cu un vârf, dacă îl are pe acesta ca vârf terminal (arcul converge spre vârf). Arcul  $(v,w)$  este incident spre interior cu vârful  $w$ .

*Incidență spre exterior*  $\Leftrightarrow$  Un arc este incident spre exterior cu un vârf, dacă îl are pe acesta ca vârf inițial (arcul pleacă din vârf). Arcul  $(v,w)$  este incident spre exterior cu vârful  $v$ .

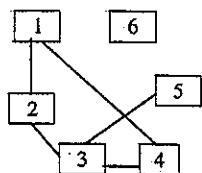
*Grad*  $\Leftrightarrow$  Gradul unui nod  $v$ , dintr-un graf neorientat, este un număr natural ce reprezintă numărul de noduri adiacente cu acesta.

*Grad interior*  $\Leftrightarrow$  În cazul unui graf orientat, fiecare nod  $v$  are asociat un număr numit grad interior și care este egal cu numărul de arce care îl au pe  $v$  ca vârf terminal (numărul de arce incidente spre interior).

*Grad exterior*  $\Leftrightarrow$  În cazul unui graf orientat, fiecare nod  $v$  are asociat un număr numit grad exterior și care este egal cu numărul de arce care îl au pe  $v$  ca vârf inițial (numărul de arce incidente spre exterior).

*Vârf izolat*  $\Leftrightarrow$  Un vârf cu gradul 0.

*Vârf terminal*  $\Leftrightarrow$  Un vârf cu gradul 1.



Vârful 5 este terminal (gradul 1).

Vârful 6 este izolat (gradul 0).

Vâfurile 1, 2, 4 au gradele egale cu 2.

*Lanț*  $\Leftrightarrow$  este o secvență de noduri ale unui graf neorientat  $G=(V,E)$ , cu proprietatea că oricare două noduri consecutive sunt adiacente:  $w_1, w_2, w_3, \dots, w_p$ , cu proprietatea că  $(w_i, w_{i+1}) \in E$  pentru  $1 \leq i < p$ .

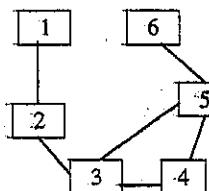
*Lungimea unui lanț*  $\Leftrightarrow$  numărul de muchii din care este format.

*Lanț simplu*  $\Leftrightarrow$  lanțul care conține numai muchii distincte.

*Lanț compus*  $\Leftrightarrow$  lanțul care nu este format numai din muchii distincte.

*Lanț elementar*  $\Leftrightarrow$  lanțul care conține numai noduri distincte.

*Ciclu*  $\Leftrightarrow$  Un lanț în care primul nod coincide cu ultimul. Ciclul este elementar dacă este format doar din noduri distincte, exceptie făcând primul și ultimul. Lungimea minimă a unui ciclu este 3.



Succesiunea de vârfuri 2, 3, 5, 6 reprezintă un lanț simplu și elementar de lungime 3.

Lanțul 5 3 4 5 6 este simplu dar nu este elementar.

Lanțul 5 3 4 5 3 2 este compus și nu este elementar.

Lanțul 3 4 5 3 reprezintă un ciclu elementar.

*Drum*  $\Leftrightarrow$  este o secvență de vârfuri ale unui graf orientat  $G=(V,E)$ , cu proprietatea că oricare două vârfuri consecutive sunt adiacente:  $(w_i, w_{i+1}), \dots, (w_p, w_1)$ , cu proprietatea că  $(w_i, w_{i+1}) \in E$ , pentru  $1 \leq i < p$ .

*Lungimea unui drum*  $\Leftrightarrow$  numărul de arce din care este format.

*Drum simplu*  $\Leftrightarrow$  drumul care conține numai arce distincte.

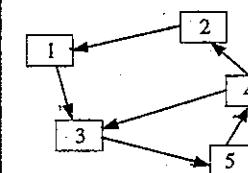
*Drum compus*  $\Leftrightarrow$  drumul care nu este format numai din arce distincte.

*Drum elementar*  $\Leftrightarrow$  drumul care conține numai vârfuri distincte.

*Circuit*  $\Leftrightarrow$  Un drum în care primul vârf coincide cu ultimul. Circuitul este elementar dacă este format doar din vârfuri distincte, exceptie făcând primul și ultimul.

*Buclă*  $\Leftrightarrow$  Circuit format dintr-un singur arc.

Ciclu elementar 3,6,4,5,1,3



Circuit elementar 1,3,5,4,2,1

*Graf parțial*  $\Leftrightarrow$  Un graf  $G'=(V,E')$  reprezintă graf parțial al grafului  $G=(V,E)$  dacă  $E' \subseteq E$ . Cu alte cuvinte  $G'$  este graf parțial al lui  $G$ , dacă este identic, sau se obține prin suprimarea unor muchii (respectiv arce) din  $G$ .

*Subgraf*  $\Leftrightarrow$  Un subgraf al lui  $G=(V,E)$  este un graf  $G'=(V',E')$  în care  $V' \subseteq V$ , iar  $V'$  conține toate muchiile/arcele din  $E$  ce au ambele extremități în  $V'$ . Cu alte cuvinte  $G'$  este subgraf al lui  $G$ , dacă este identic, sau se obține prin suprimarea unor noduri împreună cu muchiile/arcele incidente cu acestea.

*Graf regulat*  $\Leftrightarrow$  graf neorientat în care toate nodurile au același grad.

*Graf complet*  $\Leftrightarrow$  graf neorientat  $G=(V,E)$  în care există muchie între oricare două noduri. Numărul de muchii ale unui graf complet este  $|V| * |V-1| / 2$ .

*Graf conex*  $\Leftrightarrow$  graf neorientat  $G=(V,E)$  în care, pentru orice pereche de noduri  $(v,w)$ , există un lanț care le unește.

*Graf sare conex*  $\Leftrightarrow$  graf orientat  $G=(V,E)$  în care, pentru orice pereche de vârfuri  $(v,w)$ , există drum de la  $v$  la  $w$  și un drum de la  $w$  la  $v$ .

*Componentă conexă*  $\Leftrightarrow$  subgraf al grafului de referință, maximal în raport cu proprietatea de conexitate (între oricare două vârfuri există lanț);

*Lanț hamiltonian*  $\Leftrightarrow$  un lanț elementar care conține toate nodurile unui graf.

*Ciclu hamiltonian*  $\Leftrightarrow$  un ciclu elementar care conține toate nodurile grafului.

*Graf hamiltonian*  $\Leftrightarrow$  un graf  $G$  care conține un ciclu hamiltonian.

*Condiție de suficiență*: Dacă  $G$  este un graf cu  $n \geq 3$  vârfuri, astfel încât gradul oricărui vârf verifică inegalitatea:  $gr(x) \geq n/2$ , rezultă că  $G$  este graf hamiltonian.

*Lanț eulerian*  $\Leftrightarrow$  un lanț simplu care conține toate muchiile unui graf.

*Ciclu eulerian*  $\Leftrightarrow$  un ciclu simplu care conține toate muchiile grafului.

*Graf eulerian*  $\Leftrightarrow$  un graf care conține un ciclu eulerian.

*Condiție necesară și suficientă*: Un graf este eulerian dacă și numai dacă oricare vârf al său are gradul par.

## 2.1.2 Moduri de reprezentare la nivelul memoriei

Există mai multe modalități standard de reprezentare a unui graf  $G=(V, E)$ :

1. matricea de adiacență;
2. liste de adiacență;
3. matricea ponderilor (costurilor);
4. lista muchiilor.

• *Matricea de adiacență* este o matrice binară (cu elemente 0 sau 1) care codifică existența sau nu a muchiei/arcului între oricare pereche de vârfuri din graf. Este indicată ca mod de memorare a grafurilor, în special în cazul grafurilor dense, adică cu număr mare de muchii ( $|V|^2 = E$ ).

```

1 Creare_MA_neorientat(G=(V,E)) /*complexitate: O(|V|^2 + E)*/
2   pentru i=1, |V| execută
3     pentru j = 1 to |V| execută G[i][j] := 0;
4   ■
5   ■
6   pentru e = 1, |E| execută
7     citeste i, j;
8     G[i][j] := 1;
9     G[j][i] := 1; //instructiunea lipseste in cazul digrafului
10  ■

```

Implementarea în limbaj a subprogramului care creează matricea de adiacență a unui graf neorientat ia în considerare următoarele declarații:

```

const MAX_N=101;
MAX_M=1001;
var n,m:longint;
G:array[0..max_n,0..max_n] of byte;
#include <stdio.h>
#define MAX_N 101
#define MAX_M 1001
int N, M; char G[MAX_N][MAX_N];

```

Subprogramul *citeste\_graf()* preia informațiile din fișierul text *graf.in*, în felul următor: de pe prima linie numărul  $n$  de vârfuri și  $m$  de muchii, iar de pe următoarele  $m$  linii, perechi de numere reprezentând muchiile grafului. Matricea  $G$  de adiacență se consideră inițializată cu valoarea 0.

```

procedure citeste_graf;
var i,j:longint;
begin
  assign(input,'graf.in');
  reset(input);
  readln(n,m);
  while m>0 do begin
    readln(i,j);
    g[i][j]:=1;g[j][i]:=1;
    dec(m);
  end;
end;

```

```

void citeste_graf(void)
{
  int i, j;
  freopen("graf.in", "r", stdin);
  scanf("%d %d", &N, &M);
  for (; M > 0; M--)
  {
    scanf("%d %d", &i, &j);
    G[i][j] = G[j][i] = 1;
  }
}

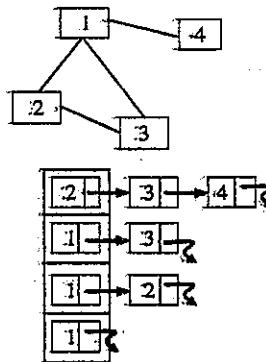
```

• *Listele de adiacență* rețin, pentru fiecare nod din graf, toate vâfurile adiacente cu acesta (vecine cu el). Ca modalitate de reprezentare se poate folosi un vector  $LA$  ale căruia elemente sunt adresele de început ale celor  $|V|$  liste simplu înălțuite memorate, una pentru fiecare vârf din  $V$ . Lista simplu înălțuită, a cărei adresă de început este reținută de  $LA[u]$  memorează toate vâfurile din  $G$ , adiacente cu  $u$  și stocate într-o ordine oarecare.

Pentru graful alăturat exemplificăm reprezentarea atât în liste de adiacență, cât și în matrice de adiacență.

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Matricea de adiacență (MA)



Liste de adiacență (LA)

Crearea listelor de adiacență este ilustrată în pseudocodul următor:

```

1 Creare_LA_neorientat(G=(V,E))      /*complexitate: O(V + E) */
2 pentru i=1, |V| execută
3   G[i] ← NIL
4   ■
5   pentru e = 1, |E| execută
6     citește i,j;
7     aloca(p);
8     p^.inf ← j; p^.leg ← G[i]; G[i] ← p
9     aloca(p);
10    p^.inf ← i; p^.leg ← G[j]; G[j] ← p
11   ■
12

```

Implementarea în limbaj a subprogramului care creează liste de adiacență a unui graf neorientat ia în considerare următoarele declarații:

```

const MAX_N=101;
type plista^=lista;
lista=record
  nod:longint;
  urm:plista;
end;
var G:array[0..max_n]of plista;
n,m,:longint;

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 101
struct lista
{
  int nod;
  lista *urm;
} *G[MAX_N];
int N, M;

```

Subprogramul *citește\_graf()* preia informațiile din fișierul text *graf.in*, în felul următor: de pe prima linie, numărul *n* de vârfuri și *m* de muchii, iar de pe următoarele *m* linii, perechi de numere reprezentând muchiile grafului. La apelul *adauga(i,j)* se realizează inserarea unui element de informație *j* în lista lui vecinilor nodului *i*. Tabloul *G*, reprezentând liste de adiacență, se consideră inițializat cu constanta *NIL*(Pascal)/*NULL* (C++).

```

1 procedure adauga(i,j:longint);
2 var p:plista;
3 begin
4   new(p); p^.nod:=j;
5   p^.urm:=G[i]; G[i]:=p;
6 end;
7
8 procedure citește_graf;
9 var i,j:longint;
10 begin
11   assign(input,"graf.in");
12   reset(input); readln(n,m);
13   while M>0 do begin
14     readln(i,j); dec(m);
15     readln(i,j); dec(m);
16     adauga(i,j); adauga(j,i);
17   end;
18 end;

```

```

void adauga(int i, int j)
{
  lista *p;
  p = new lista; p->nod = j;
  p->urm = G[i]; G[i] = p;
}

void citește_graf(void)
{
  int i, j;
  freopen("graf.in","r", stdin);
  scanf("%d %d", &N, &M);
  for (; M > 0; M--)
  {
    scanf("%d %d", &i, &j);
    adauga(i, j); adauga(j, i);
  }
}

```

Un potențial dezavantaj al reprezentării sub formă listelor de adiacență este modul oarecum anevoie de a determina dacă o muchie  $(u,v)$  este prezentă sau nu în graf. Eficiența reprezentării unui graf este dată de densitatea acestuia, deci matricea de adiacență este viabilă dacă numărul muchiilor este aproximativ egal cu  $|V|^2$ .

- Matricea costurilor este o matrice cu  $|V|$  linii și  $|V|$  coloane, care codifică existența sau nu a muchiei/arcului, între oricare pereche de vârfuri din graf, prin costul acestia. Astfel:

$$\begin{aligned} G[i, j] &= \infty, \text{ dacă } (i, j) \notin E \\ G[i, j] &= cost < \infty, \text{ dacă } (i, j) \in E \\ G[i, j] &= 0, \text{ dacă } i=j \end{aligned}$$

```

1 Creare_MC_neorientat(G=(V,E))      /*complexitate: O(V2 + E) */
2 pentru i=1, |V| execută
3   pentru j = 1 to |V| execută
4     daca i = j atunci G[i][j] ← 0;
5     altfel G[i][j] ← ∞
6   ■
7
8
9   pentru e = 1, |E| execută
10    citește i, j, c;
11    G[i][j] ← c;
12    G[j][i] ← c;           //instructiunea lipseste in cazul digrafului
13

```

Implementarea în limbaj a subprogramului care creează matricea costurilor unui graf neorientat ia în considerare următoarele declarații:

```

const MAX_N=101;
const inf=maxint div 2;
var G:array[1..MAX_N,1..MAX_N]of
  longint;
n,m,:longint;

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 101
#define INF 0x3f3f3f
int N, M, G[MAX_N][MAX_N];

```

Subprogramul `citeste_graf()` preia informațiile din fișierul text `graf.in` în felul următor: de pe prima linie, numărul  $n$  de vârfuri și  $m$  de muchii, iar de pe următoarele  $m$  linii, câte trei numere  $i, j, c$  având semnificația muchie între  $i$  și  $j$  de cost  $c$ .

```

1 procedure citeste_graf;
2 var i,j,c:longint;
3 begin
4 assign(input,'graf.in');
5 reset(input);
6 readln(n,m);
7 for i:=1 to n do
8   for j:=1 to n do
9     if i>j then G[i,j]:=inf
10    else G[i,j]:=0;
11 while m>0 do begin
12   readln(i,j,c);
13   G[i,j]:=c; G[j,i]:=c;
14   dec(m);
15 end; end;
16
17 void citeste_graf(void)
18 {
19   int i, j, c;
20   freopen("graf.in", "r", stdin);
21   scanf("%d %d", &N, &M);
22   for (i = 1; i <= N; i++)
23     for (j = 1; j <= N; j++)
24       G[i][j] = (i != j) * INF;
25   for (; M > 0; M--)
26   {
27     scanf("%d %d %d", &i, &j, &c);
28     G[i][j] = c;
29     G[j][i] = c;
30   }
31 }
```

- *Lista muchiilor* este utilizată în cadrul unor algoritmi, ea memorând, pentru fiecare muchie, cele două vârfuri incidente și eventual, costul ei.

Implementarea în limbaj a subprogramului care creează lista muchiilor unui graf neorientat ponderat, ia în considerare următoarele declarații:

```

const MAX_N=101;
MAX_M=10001;
var
n,m:longint;
E:Array[0..max_m,0..2]of longint;
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_N 101
#define MAX_M 10001
int N, M, E[MAX_M][3];
```

Tabloul  $E$  reține lista muchiilor. Subprogramul `citeste_graf()` preia informațiile din fișierul text `graf.in` în felul următor: de pe prima linie, numărul  $n$  de vârfuri și  $m$  de muchii, iar de pe următoarele  $m$  linii, câte trei numere  $i, j, c$  având semnificația muchie între  $i$  și  $j$  de cost  $c$ .

```

1 procedure citest_graf;
2 var i,j,k,c:longint;
3 begin
4 assign(input,'graf.in');
5 reset(input);
6 readln(n,m);
7 for k:=0 to M-1 do begin
8   readln(i,j,c); E[k][0]:= i;
9   E[k][1]:= j; E[k][2]:= c;
10 end;
11 end;
12
13 void citeste_graf(void)
14 {
15   int i, j, k, c;
16   freopen("graf.in", "r", stdin);
17   scanf("%d %d", &N, &M);
18   for (k = 0; k < M; k++) {
19     scanf("%d %d %d", &i, &j, &c);
20     E[k][0] = i; E[k][1] = j;
21     E[k][2] = c;
22   }
23 }
```

## 2.2. Grafuri orientate și neorientate

### 2.2.1 Teste cu alegere multiplă și duală

1. Considerăm un graf neorientat cu 8 noduri și 6 muchii, format din două componente conexe. Care este gradul maxim pe care îl poate avea un nod?

- a) 5;      b) 6;      c) 4;      d) 3.

2. Considerăm un graf neorientat cu  $n$  noduri. Care este numărul minim de muchii care se pot plasa în graf, astfel încât acesta să nu conțină noduri izolate?

- a)  $\lfloor n/2 \rfloor$ ;      b)  $n$ ;      c)  $n-1$ ;      d)  $\lfloor (n+1)/2 \rfloor$ .

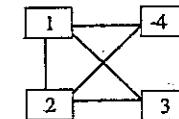
3. Considerăm un graf neorientat cu 8 noduri și 10 muchii. Care este numărul maxim de noduri terminale care pot exista în graf?

- a) 4;      b) 3;      c) 2;      d) 5.

4. Într-un graf neorientat cu 20 de noduri și 30 de muchii, numărul maxim de noduri izolate este:

- a) 10;      b) 12;      c) 11;      d) 13.

5. Considerăm graful din figura alăturată. Câte grafuri parțiale conexe formate din 3 muchii se pot obține?



- a) 10;      b) 7;      c) 6;      d) 8.

6. Fie graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=8$  și  $E=\{(1,3), (1,8), (3,8), (6,5), (6,7)\}$ . Identificați care dintre mulțimile de muchii următoare adăugate la graful  $G$  conduc la obținerea unui graf conex:

- a)  $\{(2,3), (2,8), (3,5)\}$ ;  
b)  $\{(2,5), (3,4), (3,2)\}$ ;  
c)  $\{(2,4), (4,7), (4,8), (5,7)\}$ ;  
d)  $\{(2,3), (2,7), (2,4)\}$ .

7. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=5$  și  $E=\{(1,5), (1,2), (5,2), (5,3), (3,4), (3,2), (2,4)\}$ . Referindu-ne la succesiunea de vârfuri 5 2 4 3 2 1, care dintre următoarele afirmații sunt adevărate?

- a) Reprezintă un circuit neelementar de lungime 3;  
b) Reprezintă un lanț elementar de lungime 5;  
c) Reprezintă un ciclu neelementar de lungime 3;  
d) Reprezintă un lanț simplu, dar neelementar de lungime 5.

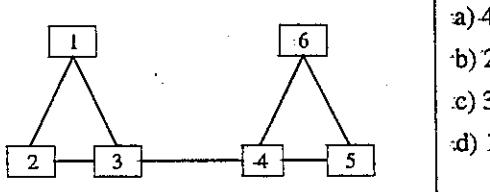
8. Fie graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=7$  și  $E=\{(1,2), (5,2), (5,3), (3,4), (3,2), (2,4)\}$ . Care sunt nodurile care nu aparțin nici unui ciclu?

- a) 1, 6, 7, 5;
- b) 6, 7, 1;
- c) 3, 6, 7;
- d) 5, 4, 6, 7.

9. Se consideră graful orientat  $G=(V, E)$ , unde  $\text{card}(V)=9$  și  $E=\{(2,1), (1,6), (2,5), (2,3), (3,4), (4,6), (5,7), (4,8), (8,9)\}$ . Identificați mulțimea nodurilor din graf ce au proprietatea că sunt legate de nodul 1 prin lanțuri a căror lungime minimă este egală cu 2.

- a) 5 4 3;
- b) 4 7;
- c) 8 7 4;
- d) 6 5 3.

10. Fie graful neorientat descris în figura următoare. Care este numărul minim de muchii care trebuie adăugate pentru ca graful să devină eulerian?



11. Se consideră graful orientat  $G=(V, E)$  unde  $\text{card}(V)=10$  și  $E=\{(1,3),(1,2), (1,6),(2,3),(4,3)(5,9),(5,7),(5,8),(6,4),(6,5),(8,2),(9,4),(9,10)\}$ . Identificați care sunt vârfurile din graf legate de vârful 1 prin drumuri de lungime minimă care trec și prin vârfurile 5 și 6 (conțin vârfurile 5 și 6):

- a) 8 4 2 9;
- b) 4 8 9 10;
- c) 8 10;
- d) 7 9 10.

12. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=6$  și  $E=\{(1,2),(1,3), (2,3),(3,4),(4,5),(4,6)\}$ . Identificați care sunt nodurile care pot fi eliminate astfel încât, subgraful cu 5 noduri rămas să fie conex:

- a) 3 4;
- b) 4 2;
- c) 1 2 5 6;
- d) 1 2 3.

13. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=6$  și  $E=\{(1,2),(1,3), (6,5),(3,4),(4,5),(4,6)\}$ . Identificați care este numărul maxim de muchii care pot fi eliminate pentru a se obține un graf parțial conex al lui  $G$ :

- a) 1;
- b) 0;
- c) 2;
- d) 3.

14. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=6$  și  $E=\{(3,4),(4,6), (3,5),(1,2),(1,3),(6,5),(2,3),(2,5),(1,4)\}$ . Identificați care este numărul minim de noduri care trebuie eliminate împreună cu muchiile incidente acestora pentru a se obține un subgraf eulerian al lui  $G$ :

- a) 0;
- b) 1;
- c) 2;
- d) 3.

15. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=12$  și  $E=\{(1,2), (1,3),(4,5),(5,6),(6,7),(4,6),(1,8)\}$ . Identificați care sunt nodurile situate în componenta conexă cu număr maxim de noduri terminale:

- a) 9 10 11 12;
- b) 4 5 6 7;
- c) 1 2 3 8;
- d) 2 3 8 7.

16. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=9$  și  $E=\{(1,6),(1,2), (1,3),(2,6),(2,3),(3,4),(3,5),(3,8)\}$ . Care este lungimea maximă a unui ciclu elementar dacă în graf se poate introduce o nouă muchie?

- a) 5;
- b) 1;
- c) 4;
- d) 3.

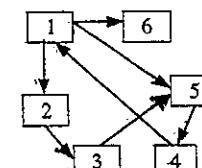
17. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=4$  și  $E=\{(1,2),(1,3), (2,3),(3,4),(1,4)\}$ . Considerând că trebuie eliminată cel puțin o muchie, câte grafuri parțiale conexe ale lui  $G$  se pot obține?

- a) 10;
- b) 13;
- c) 15;
- d) 8.

18. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=6$  și  $E=\{(1,6), (2,3),(2,5),(3,4),(4,6),(4,5)\}$ . Identificați care este numărul minim de muchii care trebuie adăugate pentru ca graful să devină regulat:

- a) 2;
- b) 3;
- c) 4;
- d) 1.

19. Se consideră graful orientat din figura următoare. Identificați care sunt vârfurile în care se poate ajunge din toate celelalte vârfuri ale grafului prin intermediul unor drumuri elementare:



20. Se consideră graful neorientat  $G=(V, E)$ , unde  $\text{card}(V)=10$  și  $E=\{(1,2), (1,3), (2,3), (2,6), (3,5), (3,6), (5,4), (5,1), (6,7), (7,3)\}$ . Identificați care este cea mai mare lungime pară a unui ciclu elementar din  $G$ :

- a) 10;      b) 4;      c) 6;      d) 8.

21. Identificați care dintre secvențele următoare reprezintă șirul gradelor nodurilor unui graf complet:

- a) 1 2 3 4;  
b) 5 5 5 5;  
c) 1 2 1 2 1 2;  
d) 4 4 4 4 4.

22. Se consideră graful neorientat conex și aciclic  $G=(V, E)$ , unde  $\text{card}(V)=7$  și  $E=\{(1,2), (2,3), (3,4), (3,5), (5,6), (6,7)\}$ . Desemnați un nod ca rădăcină pentru ca arborele astfel obținut să aibă înălțime minimă:

- a) 5;      b) 4;      c) 7;      d) 3.

23. Se consideră graful orientat  $G=(V, E)$ , unde  $\text{card}(V)=5$  și  $E=\{(1,2), (4,1), (2,3), (2,5)\}$ . Identificați numărul minim de arce care trebuie adăugate pentru ca orice vîrf să aibă gradul interior egal cu gradul exterior:

- a) 0;      b) 4;      c) 3;      d) 1.

24. Care dintre următoarele afirmații sunt adevărate?

- a) Într-un graf conex aciclic cu  $n$  noduri, numărul de muchii este  $n(n-1)/2$ ;  
b) Orice lanț compus este neelementar;  
c) Într-un graf orientat, orice vîrf are gradul exterior egal cu cel interior;  
d) Un ciclu elementar are lungimea  $> 2$ .

25. Realizați asocieri corecte între noțiunile dispuse în coloanele următoare:

- |  |  |
|--|--|
| a) lanț hamiltonian într-un graf cu $n$ noduri;      | 1) $n-1$ ;                                 |
| b) lanț eulerian într-un graf complet de $n$ noduri; | 2) lanț elementar de lungime $n-1$ ;       |
| c) suma gradelor unui graf complet cu $n$ noduri;    | 3) lanț elementar de lungime $n*(n-1)/2$ ; |
| d) numărul de muchii al unui arbore cu $n$ noduri.   | 4) $(n-1)*n$ .                             |

26. Completați următoarele afirmații:

- a) Într-un graf complet, gradul oricărui nod este egal cu.....;  
b) Suma gradelor nodurilor unui graf neorientat este egală cu dublul.....;  
c) Într-un graf neorientat orice vîrf.....are gradul 0;  
d) În orice graf neorientat există un număr.....de noduri cu grad impar.

## 2.2.2 Probleme rezolvate

### 1. Parcurgerea grafurilor în adâncime DF - Depth First

Fie graful  $G=(V, E)$ , unde  $V$  este mulțimea nodurilor și  $E$  mulțimea muchiilor. Realizați un subprogram care să permită afișarea nodurilor în cazul parcurgerii în adâncime-DF.

Soluție:

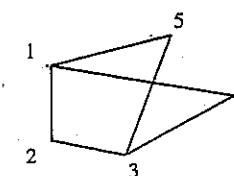
Printre operațiile fundamentale efectuate asupra structurilor de date se numără și traversarea acestora. Această operație constă într-o căutare efectuată asupra tuturor elementelor ei. Cu alte cuvinte, traversarea poate fi privită și ca un caz special de căutare, deoarece constă în regăsirea tuturor elementelor structurii.

Strategia parcurgerii în adâncime a unui graf neorientat presupune traversarea unei muchii incidente în vîrful curent, către un vîrf adjacent nedescoperit. Când toate muchiile vîrfului curent au fost explorate, se revine în vîrful care a condus la explorarea nodului curent. Procesul se repetă până în momentul în care toate vîrfurile au fost explorate.

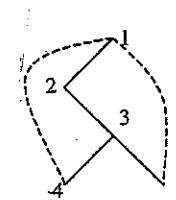
Această strategie a parcurgerii DF funcționează respectând mecanismul LIFO. Vîrful care este eliminat din stivă nu mai are nici o muchie disponibilă pentru traversare. Aceleași reguli se respectă și la parcurgerea în adâncime a grafurilor orientate (digrafuri).

În procesul de parcurgere, muchiile unui graf neorientat se pot împărti în:

- muchii de arbore (avans), folosite pentru explorarea nodurilor; ele constituie un graf parțial format din unul sau din mai mulți arbori ai parcurgerii DF.
- muchii de întoarcere (înapoi), care unesc un nod cu un strămoș al său în arborele DF.



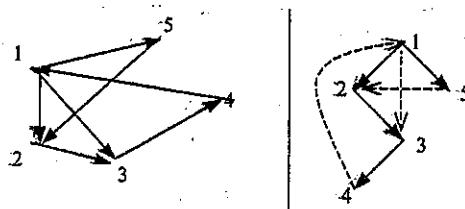
Parcurgerea DF: 1 2 3 4 5



Muchii de întoarcere: (1,4), (1,5)

În cazul digrafurilor (grafurilor orientate), se disting următoarele grupe de arce:

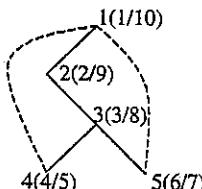
- **arce de arbore**, folosite pentru explorarea vârfurilor; ele constituie un graf parțial format din unul sau din mai mulți arbori ai parcurgerii *DF*.
- **arce înapoi**, care unește un nod cu un strămoș al său în arborele *DF*.
- **arce înainte**, sunt arce care nu aparțin arborelui *DF* și care conectează un vârf cu un descendental său.
- **arce transversale**, sunt arce care conectează două vârfuri ce nu se află în relația ascendent-descendent.



Arce transversale:  
(5,2)  
Arce înainte:(1,3)  
Arce înapoi: (4,1)  
Arce de arbore:  
(1,2), (1,5) (2,3) (3,4).

În implementarea recursivă a parcurgerii *DF*, prezentată în continuare, se folosesc următoarele tablouri unidimensionale:

- *T[N]*, în care, pentru fiecare vârf, se va reține vârful predecesor (părinte) în parcurgere;
- *D[N]*, în care, pentru fiecare vârf, se memorează momentul "descoperirii", moment care coincide cu momentul de introducere în stivă;
- *F[N]*, vector în care va memora momentul de "finish" în explorare, care coincide cu momentul extragerii din stivă;
- *Use[N]*, vector în care se codifică, în timpul parcurgerii, starea unui nod: vizitat sau nevizitat.



Valorile vectorilor:

$$\begin{aligned} D &= (1, 2, 3, 4, 6) \\ F &= (10, 9, 8, 5, 7) \\ T &= (0, 1, 2, 3, 3) \end{aligned}$$

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea nodurilor și  $E$  mulțimea muchiilor. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$  cardinalul lui  $E$ .

```

1 Depth_First (G=(V,E), D[N], F[N], T[N])    //complexitate O(N+M)
2 Use ← False;                                //nici un nod nu este vizitat
3 timp ← 0;
4 pentru nod ← 1,N execută
5   daca NOT(Use[nod]) atunci Parcurge_df(nod)
6   :
7   :
8 Parcurge_df(nod)
9 Use[nod] ← TRUE; timp ← timp+1; D[nod]←timp;
10 scrie nod; i ← prim_vecin(nod);
11 cat timp lista_vecinilor(nod)≠0 execută
12   daca NOT(Use[i]) atunci
13     T[i]←nod;
14     parcurge_df(i);
15   :
16   i ← urmator_vecin(nod);
17   :
18 timp ← timp+1; F[nod] ← timp;

```

Implementarea în limbaj a subprogramului ce realizează parcurgerea în adâncime a unui graf, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista:^lista;
lista=record
  nod:integer;
  urm:plista;
end;

var G:array[0..max_n]of plista;
T,D,F:array[0..max_n]of integer;
n,m,timp:integer;
U:array[0..max_n]of byte;

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 1001

struct lista
{
  int nod;
  lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], D[MAX_N],
F[MAX_N], timp;
char U[MAX_N];

```

Ca și în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacență (vezi 2.1.2).

```

1 procedure df(nod:integer);
2 var p:plista;
3 begin
4   U[nod]:=1;inc(timp);
5   D[nod]:=timp;
6   write(nod, ' '); p:=G[nod];
7   while p>>nil do begin
8     if U[p^.nod]=0 then begin
9       T[p^.nod]:=nod;
10      DF(p^.nod);
11    end;
12    p:=p^.urm;
13  end;
14  inc(timp); F[nod]:=timp;
15 end;

```

```

void DF(int nod)
{
  lista *p;
  U[nod] = 1;
  D[nod] = ++timp;
  printf("%d ", nod);
  for (p = G[nod]; p != NULL;
  p = p->urm)
  if (!U[p->nod])
  {
    T[p->nod] = nod;
    DF(p->nod);
  }
  F[nod] = ++timp;
}

```

## 2. Parcugerea grafurilor în lățime BF - Breadth First

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea nodurilor și  $E$  mulțimea muchiilor. Realizați un subprogram care să permită afișarea nodurilor în cazul parcugeriîi în lățime-BF.

### Solutie:

Strategia parcugeriîi BF funcționează respectând mecanismul de tip FIFO. Ea se bazează pe traversarea tuturor muchiilor disponibile din nodul curent către noduri adiacente nedescoperite, care vor fi astfel vizitate. După acest proces, nodul explorat este scos din coadă, prelucrându-se succesiv toate nodurile ajunse în vârful cozii.

Acest mecanism permite identificarea lanțurilor de lungime minimă de la nodul de start către toate vârfurile accesibile lui din graf. Arborele BF, ce cuprinde muchiile traversate în parcugere în lățime, are proprietatea de a fi format doar din lanțuri de lungime minimă, lanțuri ce unesc nodul de start al parcugeriîi cu toate vârfurile accesibile acestuia.

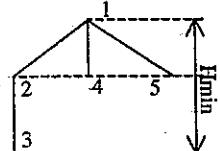
Aceleași reguli se respectă și la parcugere în lățime a grafurilor orientate (digrafuri).

Algoritmul lui Lee de determinare a lanțului minim dintre două vârfuri ale unui graf se bazează tot mai pe această strategie de traversare în lățime, nodul de plecare fiind unul dintre cele două vârfuri date.

În implementarea iterativă a parcugeriîi BF, prezentată în continuare, se folosesc următoarele tablouri unidimensionale:

- $T[N]$ , în care, pentru fiecare vârf, se va reține vârful predecesor (părinte) în parcugere;
- $D[N]$ , în care, pentru fiecare vârf, se memorează lungimea lanțului minim către nodul de start;
- $C[N]$ , vector (coadă) în care se memora ordinea de parcugere a nodurilor în BF;
- $Use[N]$ , vector în care se codifică, în timpul parcugeriîi, starea unui nod: vizitat sau nevizitat.

Pentru graful considerat anterior ca exemplu, arborele BF este următorul:



Valorile vectorilor:

$$\begin{aligned} C &= (1, 2, 4, 5, 3) \\ D &= (0, 1, 2, 1, 1) \\ T &= (0, 1, 2, 1, 1) \end{aligned}$$

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea nodurilor și  $E$  mulțimea muchiilor. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$  cardinalul lui  $E$ .

```

1 | Breadth_First : (G:=(V,E), C[N], T[N], Nod_Start)           /*complexitate: O(M+N)*/
2 |
3 | Use ← FALSE;
4 | Coada ← {Nod_Start}    //nodul de start este introdus in coada
5 | Use[nod] ← True;
6 | cat_timep Coada ≠ φ execută
7 |   v ← varf_Coada; i ← prim_vecin(v);
8 |   cat_timep lista_vecinilor(v) ≠ φ execută
9 |     dacă not(Use[i]) atunci
10|       T[i] ← v; Use[i] ← TRUE;
11|       D[i] ← D[v]+1;
12|       Coada ← {i};
13|
14|     i ← urmator_vecin(v);
15|
16| COADA ← v;
17|

```

Implementarea în limbaj a subprogramului ce realizează parcugerea în lățime a unui graf, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista^=lista;
  lista=record
    nod:integer; urm:plista;
  end;
var G:array[0..max_n]of plista;
T,D,C:array[0..max_n]of integer;
n,m,timp:integer;
U:Array[0..max_n]of byte;

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 1001
struct lista
{
  int nod;
  lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], D[MAX_N],
C[MAX_N], timp; char U[MAX_N];

```

Ca și în pseudocod, s-a preferat implementarea în manieră iterativă, iar graful este considerat a fi memorat cu ajutorul listelor de adiacență (vezi 2.1.2).

```

1 | procedure BF(start:integer);
2 | var p:plista;
3 |   nod,st,dr:integer;
4 | begin
5 |   fillchar(U,sizeof(u),0);
6 |   st:=1; dr:=1; C[st]:=start;
7 |   U[start]:=1; D[start]:=0;
8 |   while st<=dr do begin
9 |     nod:=C[st]; p:=G[nod];
10|    while p<=nil do begin
11|      if U[p^.nod]=0 then begin
12|        inc(dr); C[dr]:=p^.nod;
13|        D[p^.nod]:=D[nod]+1;
14|        u[C[dr]]:=1; T[p^.nod]:=nod;
15|      end;
16|      p:=p^.urm;
17|    end;
18|    inc(st);
19|  end;
20| end;

```

```

void BF(int start)
{
  lista *p;
  int nod, st, dr;
  memset(U, 0, sizeof(U));
  st = dr = 1;
  C[st]=start;
  U[start] = 1;
  for (D[start]=0;st<=dr;st++)
  {nod=C[st];
    for (p = G[nod];p != NULL;
      p = p->urm)
      if (!U[p->nod])
      {
        U[C[++dr] = p->nod] = 1;
        D[p->nod] = D[nod]+1;
        T[p->nod] = nod;
      }
  }
}

```

### 3. Ciclu eulerian

Fie  $G=(V,E)$ , graf neorientat conex, în care fiecare nod are gradul par. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$ , cardinalul lui  $E$ . Să se determine un ciclu eulerian al grafului  $G$ , altfel spus, un ciclu simplu de lungime  $M$ .

#### Exemplu:

Considerând graful  $G$  în care  $N=6$ ,  $M=10$  și arcele: (1,2), (1,3), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5), (4,6), (5,6), se va afișa: 1,3,5,6,4,5,2,4,3,2,1.

#### Solutie:

Pentru a construi un ciclu eulerian se va parcurge graful folosind o strategie asemănătoare cu cea a parcurgerii în adâncime a unui graf, strategie care respectă mecanismul *LIFO*.

Înaintarea către un vîrf adiacent vîrfului curent se va face simultan cu eliminarea muchiei respective. În acest fel, nodurile nu vor mai fi marcate (ca la parcurgerea *DF*) pentru a putea fi vizitate de mai multe ori.

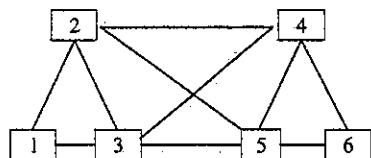
De fiecare dată când nodul curent este eliminat din stivă, acesta este afișat sau salvat într-o coadă. Această coadă va reține, în ordine, nodurile ciclului eulerian. Procesul se repetă până în momentul în care stiva devine vidă, deci toate muchiile au fost traversate.

```

1  ciclu_euler (Nod)           /*complexitate: O(M+N)*/
2
3  i ← prim_vecin(nod);
4  cat timp lista_vecinilor(nod) ≠ φ execută
5      elimin(i,nod)           //elimin pe i din vecinii lui nod
6      elimin(nod,i)          //elimin pe nod din vecinii lui i
7      ciclu_euler(i)
8      i ← urmator_vecin(nod);
9
10 Coada ← {nod};

```

Considerăm graful următor și desemnăm ca nod de start pentru construirea ciclului eulerian, nodul 1:



#### Pasul 1:

Se parcurge începând cu nodul 1

Stiva = (1,2,3,1)

Coada =  $\emptyset$

#### Pasul 2:

Iese din stivă nodul 1, salvându-se în coadă;

Stiva = (1,2,3)

Coada = (1)

#### Pasul 3:

Se continuă parcurgerea:

Stiva = (1,2,3,4,2,5,3)

Coada = (1)

#### Pasul 4:

Iese din stivă nodul 3, salvându-se în coadă;

Stiva = (1,2,3,4,2,5)

Coada = (1,3)

#### Pasul 5:

Se continuă parcurgerea:

Stiva = (1,2,3,4,2,5,4,6,5)

Coada = (1,3)

#### Pasul 6:

Iese din stivă nodul 5, salvându-se în coadă;

Stiva = (1,2,3,4,2,5,4,6)

Coada = (1,3,5)

#### Pasul 7:

Toate nodurile sunt extrase din stivă și introduse în coadă. La finalul algoritmului:

Stiva =  $\emptyset$

Coada = (1,3,5,6,4,5,2,4,3,2,1).

Implementarea în limbaj a subprogramului ce realizează determinarea unui ciclu eulerian, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=101;
MAX_M=1001;
var C:array[0..max_m]of integer;
    n,m,nc:integer;
G :array[0..max_n,0..max_n]of byte;

```

```

#include <stdio.h>
#define MAX_N 101
#define MAX_M 1001
int N, M, C[MAX_N], nc;
char G[MAX_N][MAX_N];

```

Ca și în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul matricei de adiacență (vezi 2.1.2). Datorită acestui mod de memorare, implementarea în limbaj conduce la o complexitate pătratică  $O(N^2)$ . Dacă graful este memorat cu ajutorul listelor de adiacență, complexitatea este redusă la  $O(N+M)$ .

```

1 procedure euler(nod:integer);
2 var urm:integer;
3 begin
4     for urm:=1 to n do
5         if G[nod,urm]<>0 then begin
6             G[nod,urm]:=0;
7             G[urm,nod]:=0;
8             euler(urm);
9         end;
10    C[nc]:=nod; inc(nc);
11 end;

```

```

void euler(int nod)
{
    int urm;
    for (urm = 1; urm <= N; urm++)
        if (G[nod][urm])
        {
            G[nod][urm] = 0;
            G[urm][nod] = 0;
            euler(urm);
        }
    C[nc++] = nod;
}

```

În cazul în care se dorește un lanț eulerian, sau un ciclu, se poate aplica același algoritm începând dintr-un nod cu grad impar, dacă există vreunul. Un lanț eulerian se poate forma numai dacă există zero sau două noduri cu grad impar.

#### 4. Matricea drumurilor - Algoritmul lui Warshall

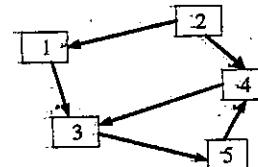
Fie  $G=(V,E)$ , graf orientat cu  $n$  vârfuri și  $m$  arce. Să se determine, pentru orice pereche de vârfuri  $x, y \in V$ , dacă există sau nu un drum format din unul sau din mai multe arce între  $x$  și  $y$ .

##### Soluție:

Numim matricea de drumuri sau a închiderii tranzitive, o matrice pătratică  $D(N,N)$  în care elementul  $D[i,j]=1$ , dacă există drum între nodul  $i$  și nodul  $j$  și 0, în caz contrar.

De exemplu, pentru graful ilustrat alăturat, matricea de drumuri este următoarea:

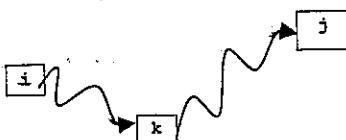
```
0 0 1 1 1  
1 0 1 1 1  
0 0 1 1 1  
0 0 1 1 1  
0 0 1 1 1
```



Algoritmul Warshall construiește matricea închiderii tranzitive  $D$ , plecând de la matricea de adiacență  $G$ .

Înțial, matricea  $D$  indică prezența drumurilor de lungime 1 între orice pereche de vârfuri adiacente.

Algoritmul parcurge de  $n$  ori matricea  $D$ , câte o dată pentru fiecare nod  $k$  al grafului. Astfel, pentru fiecare nod  $k$ , între orice pereche de noduri  $i$  și  $j$  din graf, fie s-a descoperit deja un drum ( $D[i,j]=1$ ), fie acesta poate fi construit prin concatenarea drumurilor de la  $i$  la  $k$  și de la  $k$  la  $j$ , dacă acestea există.



Luând în considerare caracteristica logică a valorilor elementelor matricei  $D$ , atunci regula descrisă mai sus poate fi exprimată astfel:  

$$D[i,j] = D[i,j] \text{ OR } (D[i,k] \text{ AND } D[k,j])$$

```
1 warshall (G=(V,E))  
2 //complexitate: O(N^3)/**/  
3 D = G;  
4 //initializare matrice drumuri  
5 for k = 1, n execute  
6   for i = 1, n execute  
7     for j = 1, n execute  
8       D[i,j] = D[i,j] OR (D[i,k] AND D[k,j])  
9  
10
```

Implementarea în limbaj a subprogramului ce realizează determinarea matricei închiderii tranzitive, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=101;  
MAX_M=1001;  
var n,m,nc:integer;  
G,D:array[0..max_n,0..max_n]of byte  
#define MAX_N 101  
#define MAX_M 1001  
int N, M, C[MAX_M], nc;  
char G[MAX_N][MAX_N], D[MAX_N][MAX_N];
```

Ca și în pseudocod, graful este considerat să fi memorat cu ajutorul matricei de adiacență (vezi 2.1.2). Complexitatea algoritmului lui Warshall este cubică  $O(N^3)$ .

```
1 procedure Warshall;  
2 var i,j,k:integer;  
3 begin  
4   for i:=1 to n do  
5     for j:=1 to n do  
6       d[i,j]:=g[i,j];  
7   for k:=1 to n do  
8     for i:=1 to n do  
9       for j:=1 to n do  
10      if (d[i,j]=0) then  
11        d[i,j]:=d[i,k] AND d[k,j];  
12  end;
```

```
void Warshall ()  
{  
  int i,j,k;  
  for (i = 1; i <= N; i++)  
    for (j = 1; j <= N; j++)  
      D[i][j] = G[i][j];  
  for (k = 1; k <= N; k++)  
    for (i = 1; i <= N; i++)  
      for (j = 1; j <= N; j++)  
        if (! D[i][j])  
          D[i][j]=D[i][k] && D[k][j];  
}
```

#### 5. Componente conexe

Fie  $G=(V,E)$  un graf neorientat. Se dorește determinarea componentei conexe cu număr maxim de noduri din  $G$ . Componenta va fi identificată printr-unul dintre vârfurile sale și numărul de vârfuri din care este formată.

*Exemplu:* Considerând graful  $G$  în care  $N=6$ ,  $M=6$  și arcele: (1,2), (3,4), (3,5), (4,5), (4,6), (5,6) se va afișa: 3 4 (nodul 3 face parte din componenta conexă formată din 4 noduri).

##### Soluție:

Problema va fi rezolvată prin determinarea tuturor componentelor conexe ale grafului și identificarea componentei cu număr maxim de noduri.

Ne reamintim că o componentă conexă este un subgraf maximal în raport cu proprietatea de conexitate.

Pentru a descompune graful în componente conexe, vom proceda în felul următor: vom realiza o parcurgere  $DF$  din nodul 1, determinându-se componenta conexă din care acestea fac parte. Vom continua algoritmul cu o nouă parcurgere efectuată dintr-un nod nevizitat anterior. Procedeul continuă până când s-au vizitat toate nodurile.

Algoritmul de descompunere în componente conexe a unui graf neorientat este prezentat în pseudocod, în continuare.

```

1 Componente_conexe (G=(V,E))           //**complexitate: O(M+N)**
2   nrc ← 0;                           // nr de componente conexe
3   Use ← False;                      // nici un nod selectat
4   pentru i ← 1,n executa
5     daca Not(Use[i]) atunci
6       nrc ← nrc + 1;
7       parcurge_df(i)
8
9

```

Pentru a identifica cea mai numeroasă componentă conexă, vom determina în cadrul fiecărei parcurgeri DF numărul de noduri selectate.

Implementarea în limbaj a subprogramelor prezentate în continuare ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista^lista;
    lista=record
      nod:integer; urm:plista;
    end;
var G:array[0..max_n]of plista;
T,D,F:array[0..max_n]of integer;
n,m,temp:integer;
U:Array[0..max_n]of byte;

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 1001
struct lista
{
  int nod; lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], D[MAX_N],
F[MAX_N], temp;
char U[MAX_N];

```

Graful este considerat a fi memorat cu ajutorul listelor de adiacență (vezi 2.1.2).

```

procedure df(nod:integer;
            var x:integer);
var p:plista;
begin
  U[nod]:=1; inc(x); p:=G[nod];
  while p<>nil do begin
    if U[p^.nod]=0 then
      DF(p^.nod,x);
    p:=p^.urm;
  end;
end;

procedure comp_conex;
var i,max,v,x:integer;
begin
  fillchar(U,n,0);
  max:=0;
  for i:=1 to n do
    if U[i]=0 then begin
      x:=0; df(i,x);
      if x>max then begin
        max:=x; v:=i end;
      end;
      writeln(max,' ',v);
    end;

```

```

void DF(int nod, int &x)
{
  lista *p;
  U[nod] = 1;
  x++;
  for (p = G[nod]; p != NULL;
       p = p->urm)
    if (!U[p->nod])
      DF(p->nod,x);
}

void comp_conex()
{
  int i,max,v,x;
  max=0;
  memset(U, 0, sizeof(U));
  for (i = 1; i <= N; i++)
    if (!U[i])
    {x=0;
      DF(i,x);
      if (x>max)
        max=x; v=i;
    }
  printf("%d %d",max,v);
}

```

## 6. Tare-conexitate

Fie  $G=(V,E)$  un graf orientat. Realizați un program care afișează vârfurile fiecărei componente tare conexe în care se descompune graful  $G$ .

*Exemplu:* Considerând digraful  $G$  în care  $N=5$ ,  $M=6$  și arcele: (1,2), (1,5), (2,3), (3,4), (3,5), (4,1), se vor afișa două componente tare conexe:

1 4 3 2

5

### Solutie:

În cazul digrafurilor (grafurilor orientate), o componentă tare conexă reprezintă un subgraf maximal în care, oricare două vârfuri sunt accesibile unul din celălalt.

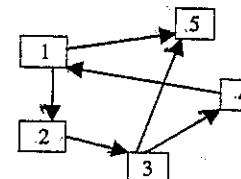
În cadrul unei componente tare conexe, inversarea sensului de deplasare nu implică blocarea accesului către vreunul din vârfurile sale.

Algoritmul propus identifică componenta tare conexă a unui vârf ca fiind intersecția dintre mulțimea nodurilor accesibile din el în graful inițial  $G$  și mulțimea nodurilor accesibile din el în graful transpus  $G_t$  (obținut prin inversarea sensurilor arcelor).

Acest algoritm se bazează pe parcurgerea DF a celor două grafuri, de aici și complexitatea liniară a acestuia  $O(N+M)$ . Operațiile efectuate sunt:

- Parcurgerea DF a grafului inițial ( $G$ ) pentru memorarea explicită a stivei ce conține vârfurile grafului în ordinea crescătoare a timpilor de finish.
- Parcurgerea DF a grafului transpus ( $G_t$ ) începând cu ultimul vârf reținut în stivă către primul. Parcurgerea se reia din fiecare vârf rămas nevizitat la parcurgerile anterioare. Vârfurile fiecărui arbore DF obținut la acest pas reprezintă câte o componentă tare conexă.

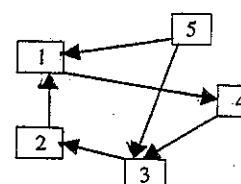
Pentru graful:



### Pasul 1:

Stiva DF cuprinzând nodurile în ordinea crescătoare a timpilor de finish este:

$St=(4, 5, 3, 2, 1)$ .



### Pasul 2:

Parcurgerea DF pe graful transpus începând cu  $St[n]$  acțează vârfurile din prima componentă tare conexă: {1,4,3,2}.

Parcurgerea DF din primul vârf rămas neselectat  $St[2]$  acțează vârfurile celei de a două componente tare conexe: {5}.

Implementarea în limbaj a subprogramelor prezentate în continuare ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista^list;
list = record
    nod:integer; urm:plista;
end;
graf=array[0..max_n]of plista;
var G,GT:graf;
ST:array[0..max_n]of integer;
i,n,m,nst:integer;
U:array[0..max_n]of byte;

```

Tabloul  $GT$  va reține graful transpus asociat lui  $G$ . Subprogramul *citeste\_graf* preia datele referitoare la graful  $G$  și construiește matricele de adiacență ale celor două grafuri  $G$  și  $GT$ .

```

1 procedure DF1(nod:integer);
2 var p:plista;
3 begin
4     U[nod]:= 1; p:=G[nod];
5     while p<>nil do begin
6         if (!U[p^.nod]=0)then
7             DF1(p^.nod);
8         p:=p^.urm;
9     end;
10    inc(nst); ST[nst]:=nod;
11 end;
12
13 procedure DF2(nod:integer);
14 var p:Plista;
15 begin
16     U[nod]:=1; write(nod, ' ');
17     p:=gt[nod];
18     while p<>nil do begin
19         if (!U[p^.nod]=0) then
20             DF2(p^.nod);
21         p:=p^.urm;
22     end;
23 end;
24
25 procedure citeste_graf;
26 begin ..... end;
27
28 begin
29     citeste_graf;
30     for i:=1 to N do
31         if u[i]=0 then DF1(i);
32     fillchar(u,sizeof(u),0);
33     for i:=n downto 1 do
34         if u[ST[i]]=0 then begin
35             DF2(ST[i]);
36             writeln;
37         end;
38     end.

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 1001

struct lista
{
    int nod;
    lista *urm;
} *G[MAX_N], *GT[MAX_N];
int n, m, ST[MAX_N], nst;
char U[MAX_N];

```

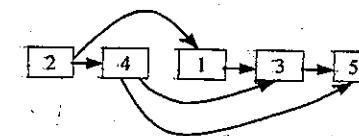
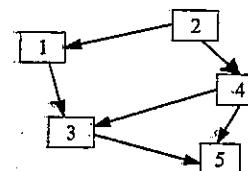
## 7. Sortarea topologică a unui digraf

Fie  $G=(V,E)$  un graf orientat aciclic. Realizați un program care ordonează vârfurile grafului după următoarea regulă: pentru orice arc  $(x,y) \in E$ , vârful  $x$  apare, în sirul ordonat, înaintea vârfului  $y$ .

*Exemplu:* Considerând digraful  $G$  în care  $N=5$ ,  $M=6$  și arcele:  $(1,3)$ ,  $(2,1)$ ,  $(2,4)$ ,  $(3,5)$ ,  $(4,3)$ ,  $(4,5)$ , se va afișa  $2\ 4\ 1\ 3\ 5$ .

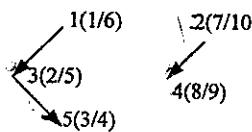
Soluție:

Pentru a înțelege mai bine modul de ordonare a vârfurilor, realizat de sortarea topologică, să urmărim graful din exemplu:



O ordonare corectă a vârfurilor este:  $2\ 4\ 1\ 3\ 5$

Presupunem că se realizează o parcurgere *DF* a grafului prezentat. Vectorii  $D$  și  $F$  care rețin timpii de intrare, respectiv timpii de ieșire din stivă sunt:



$$D=(1, 7, 2, 8, 3) \\ F=(6, 10, 5, 9, 4)$$

Dacă vârfurile sunt reținute într-o listă, în ordinea ieșirii lor din stivă, atunci această listă va conține:  $ST=(5, 3, 1, 4, 2)$ . Afisarea conținutului ei în ordine inversă, adică în ordinea inversă a timpilor de finish, va reprezenta ordinea vârfurilor obținută prin sortarea topologică  $2, 4, 1, 3, 5$ .

Să presupunem vârfurile digrafului ca niște evenimente și fiecare arc  $(x,y)$  considerăm că indică faptul că evenimentul  $x$  trebuie să se execute înaintea evenimentului  $y$ . Plecând de la aceste considerante, deducem că sortarea topologică induce o ordine asupra evenimentelor, astfel încât acestea să poată fi executate.

Să asociem următoarele evenimente grafului anterior: Nodul 1 - Mă îmbrac; Nodul 2 - Mă trezesc; Nodul 3 - Servesc masa; Nodul 4 - Mă spăl; Nodul 5 - Plec din casă. Atunci ordinea evenimentelor dată de sortarea topologică este: Mă trezesc, mă spăl, mă îmbrac, servesc masa, plec din casă.

Fie graful  $G=(V,E)$ , unde  $V$  este mulțimea vârfurilor și  $E$ , mulțimea arcelor. Notăm cu  $N$  cardinalul lui  $V$  și cu  $M$ , cardinalul lui  $E$ .

```

1 Sortare_topologică_DF (G=(V,E),St[N]) //complexitate O(N+M)
2 nr ← 0; //nr de vârfuri extrase din stivă
3 Use ← False
4 pentru nod ← 1..N execută
5   dacă Not(Use[nod]) atunci Parcurge_df(nod)
6   ■
7   ■
8   pentru i ← N,1,-1 execută scrie St[i]
9   ■
10 Parcurge_df(nod)
11 Use[nod] ← TRUE; i ← prim_vecin(nod);
12 cat_timp lista_vecinilor(nod)≠# execută
13   dacă not(Use[i]) atunci parcurge_df(i);
14   ■
15   i ← urmator_vecin(nod);
16   ■
17 nr ← nr+1; St[nr] ← nod;

```

Implementarea în limbaj a subprogramului ce realizează sortarea topologică a unui digraf aciclic, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista^lista;
  lista=record
    nod:integer;urm:Plista;end;
var G:Array[0..max_n]of plista;
  n,i,m,nst:integer;
  st:array[0..max_n]of integer;
  u:array[0..max_n]of byte;
  #include <stdio.h>
  #define MAX_N 1001
  struct lista
  {
    int nod;
    lista *urm;
  } *G[MAX_N];
  int N, M, ST[MAX_N], nst;
  char U[MAX_N];

```

Ca și în pseudocod, s-a preferat implementarea în manieră recursivă, iar graful este considerat a fi memorat cu ajutorul listelor de adjacență (vezi 2.1.2).

```

1 procedure df(nod:integer);
2 var p:plista;
3 begin
4   u[nod]:=1; p:=G[nod];
5   while p<>nil do begin
6     if u[p^.nod]=0 then
7       Df(p^.nod);
8     p:=p^.urm;
9   end;
10 inc(nst);
11 St[nst]:=nod;
12 end;
13
14 begin
15   citeste_graf();
16   for i:=1 to N do
17     if u[i]=0 then Df(i);
18   for i:=n downto 1 do
19     write(st[i],' ');
20 end.
  void DF(int nod)
  {
    lista *p;
    U[nod] = 1;
    for (p = G[nod]; p != NULL;
         p = p->urm)
      if (!U[p->nod])
        DF(p->nod);
    ST[nst++] = nod;
  }

  int main(void)
  {
    int i;
    citeste_graf();
    for (i = 1; i <= N; i++)
      if (!U[i]) DF(i);
    for (i = N-1; i >= 0; i--)
      printf("%d ", ST[i]);
    return 0;
  }

```

O altă modalitate de implementare a sortării topologice ține cont de observația că, la un moment dat, un eveniment poate fi executat, dacă nu există nici un alt eveniment de care acesta depinde, care să nu fi fost executat anterior.

Revenind la modelarea pe digrafuri, gradul interior al unui vârf reprezintă tot mai numărul de evenimente (vârfuri) de care acesta depinde.

Înțial, în graf trebuie identificate vârfurile cu gradul interior nul, ele fiind plasate primele într-o coadă care va reține, la finalul algoritmului, ordinea vârfurilor date de sortarea topologică. Vom parcurge graful în lățime, pornind din primul vârf plasat în coadă. La fiecare pas, vom decrementa gradele tuturor vârfurilor adiacente spre interior cu acestea. În coadă vor fi adăugate doar vârfurile vecine cu cel curent, neselectate anterior și care au gradul interior nul. Algoritmul se încheie când toate vârfurile au fost plasate în coadă.

Pentru o mai bună înțelegere, să privim exemplul următor în care vectorul  $Deg(N)$  reține gradele interioare ale vârfurilor, iar coada  $C(N)$  va memora vârfurile în ordinea indușă de sortarea topologică:

Înțial vectorii conțin:

$$Deg=(1,0,2,1,2) \\ C=(2)$$

a) Se parcurge în lățime din nodul 2

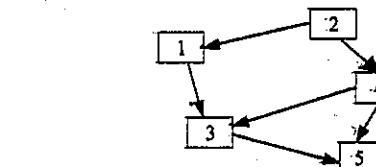
$$Deg=(0,0,2,0,2) \\ C=(2,4,1)$$

b) Se parcurge în lățime din nodul 4

$$Deg=(0,0,1,0,1) \\ C=(2,4,1)$$

c) Se parcurge în lățime din nodul 1

$$Deg=(0,0,0,0,1) \\ C=(2,4,1,3)$$



e) Se parcurge în lățime din nodul 3  
 $Deg=(0,0,0,0,0)$   
 $C=(2,4,1,3,5)$

Implementarea în limbaj a subprogramului ce realizează sortarea topologică a unui digraf aciclic, folosind parcurgerea  $BF$ , prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista^lista;
  lista=record
    nod:integer;urm:Plista;end;
var G:Array[0..max_n]of plista;
  C,Deg:Array[0..max_n]of integer;
  n,m:integer;
  u:array[0..max_n]of byte;
  #include <stdio.h>
  #define MAX_N 10001
  struct lista
  {
    int nod;
    lista *urm;
  } *G[MAX_N];
  int N, M, C[MAX_N], deg[MAX_N];
  char U[MAX_N];

```

Graful este considerat a fi memorat cu ajutorul listelor de adiacență (vezi 2.1.2).

```

1 procedure sort_bf;
2 var p:plista;
3 i,st,dr:integer;
4 begin
5 st:=0;dr:=-1;
6 for i:=1 to n do
7 if deg[i]=0 then begin
8 inc(dr);C[dr]:=i;U[i]:=1 end;
9 while st<=dr do begin
10 p:=G[C[st]];
11 while p<>nil do begin
12 dec(deg[p^.nod]);
13 if (u[p^.nod]=0)and
14 (deg[p^.nod]=0) then begin
15 inc(dr); C[dr]:=p^.nod;
16 u[p^.nod]:=1;
17 end;
18 p:=p^.urm;
19 end;
20 inc(st);
21 end;
22 end;

```

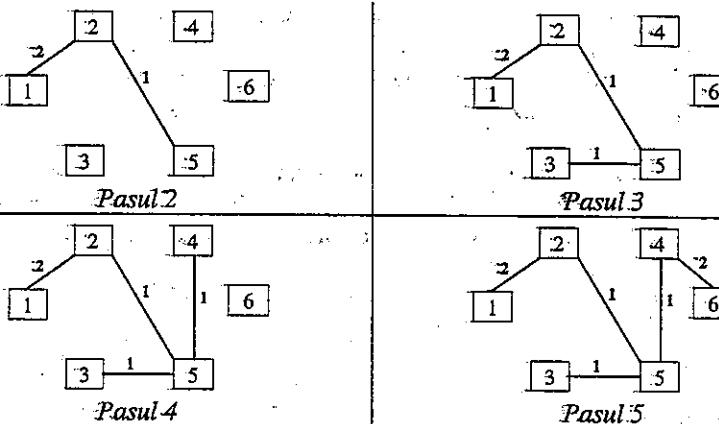
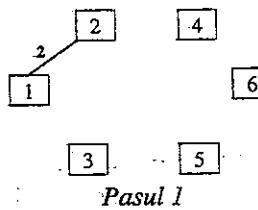
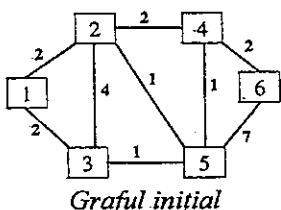
### 8. Arboare parțial de cost minim (A.P.M) – Algoritmul lui Prim

Fie  $G=(V,E)$  un graf neorientat conex, cu costuri asociate muchiilor. Un arboare parțial al lui  $G$  este un graf parțial conex și fără cicluri. Realizați un program care determină un arboare parțial de cost minim, adică un arboare parțial pentru care suma costurilor tuturor muchiilor sale este minimă.

#### Solutie:

Algoritmul lui Prim construiește arboarele parțial de cost minim, pornind de la un nod oarecare considerat rădăcină. La fiecare pas al algoritmului, la arbore se va adăuga un nou vârf. Acesta are proprietatea că este conectat de unul dintre vârfurile prezente deja în arbore, printr-o muchie de cost minim.

Să privim modul de construcție al A.P.M. cu ajutorul algoritmului lui Prim pe graful următor, considerând ca rădăcină nodul 1:



S-a obținut un A.P.M plecând de la nodul 1, costul acestuia fiind 7.

Transcrierea în pseudocod a algoritmului folosește următoarele structuri de date:

- tabloul  $D$ , în care elementul  $D[x]$  va reține costul minim prin care putem „lega” nodul  $x$ , neconectat la arbore, de orice nod al arborelui.
- tabloul  $T$ , în care elementul  $T[x]$  va reține nodul din arbore de care nodul  $x$  a fost conectat cu costul  $D[x]$ .
- Lista  $Q$  conține pe tot parcursul algoritmului nodurile grafului  $G$  neconectate la arbore.

```

1 Prim (G=(V,E,cost), rad)           //complexitate O(N*N)
2   Q ← V;                         //lista Q conține initial toate nodurile din G
3   D ← ∞;
4   D [rad] ← 0; T [rad] = 0;
5   cat timp (Q ≠ Ø) executa
6     x ← minim (Q) ;
7     Q ⇒ {x}
8     pentru fiecare y ∈ Q, adiacent cu x executa
9       daca (cost [x,y] < d [y]) atunci
10        T [y] = x;
11        D [x] = cost [x,y];
12
13
14

```

Implementarea în limbaj a algoritmului lui Prim, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=101;
INF=maxint div 2;
var n,m,r,cost:integer;
C:array[0..max_n,0..max_n]of
integer;
D,T:array[0..max_n]of integer;
U:Array[0..max_n]of byte;

```

Graful este memorat cu ajutorul matricei costurilor (vezi 2.1.2). Parametrul  $x$  indică nodul desemnat ca rădăcină a arborelui parțial de cost minim. Lista  $Q$  a fost codificată cu ajutorul vectorului  $Use$ , astfel  $Use[i]=1$ , dacă nodul  $i$  aparține arborelui și 0, în caz contrar.

Vectorul  $D$  s-a inițializat cu valorile plasate pe linia  $x$  a matricei ponderilor deoarece, la prima iterație a algoritmului, arcele minime prin care un vârf din graf poate fi conectat la rădăcina  $x$  sunt chiar costurile arcelor ce pleacă din  $x$ .

```

1 procedure prim(x:integer);
2 var i,min,nod:integer;
3 begin
4   fillchar(u,sizeof(u),0);
5   fillchar(t,sizeof(t),0);
6   for i:=1 to n do begin
7     D[i]:=c[x,i];
8     T[i]:=x;
9   end;
10  u[x]:=1;
11
12  while true do begin
13    min:=INF;
14    nod:=-1;
15    for i:=1 to n do
16      if (U[i]=0)and(min>D[i])then
17        begin
18          min:=D[i];
19          nod:=i;
20        end;
21    if (min = INF) then break;
22
23    U[nod]:=1;
24    inc(cost,D[nod]);
25    writeln(T[nod],',',nod);
26
27    for i:=1 to n do
28      if (D[i] > C[nod,i]) then
29        begin
30          D[i]:=C[nod,i];
31          T[i]:=nod;
32        end;
33    writeln(cost)
34  end;
35
```

```

void prim(int x)
{
  int i, min, nod;

  memset(U, 0, sizeof(U));
  memset(T, 0, sizeof(T));

  for (i = 1; i <= N; i++)
    D[i] = C[x][i], T[i] = x;
  U[x] = 1;

  while (1)
  {
    min = INF; nod = -1;
    for (i = 1; i <= N; i++)
      if (!U[i] && min > D[i])
        min = D[i], nod = i;
    if (min == INF) break;

    U[nod] = 1;
    cost += D[nod];
    printf("%d %d\n", T[nod], nod);

    for (i = 1; i <= N; i++)
      if (D[i] > C[nod][i])
        {
          D[i] = C[nod][i];
          T[i] = nod;
        }
    printf("%d\n", cost);
  }
}
```

#### 9. Arbore parțial de cost minim (A.P.M) – Algoritmul lui Kruskal

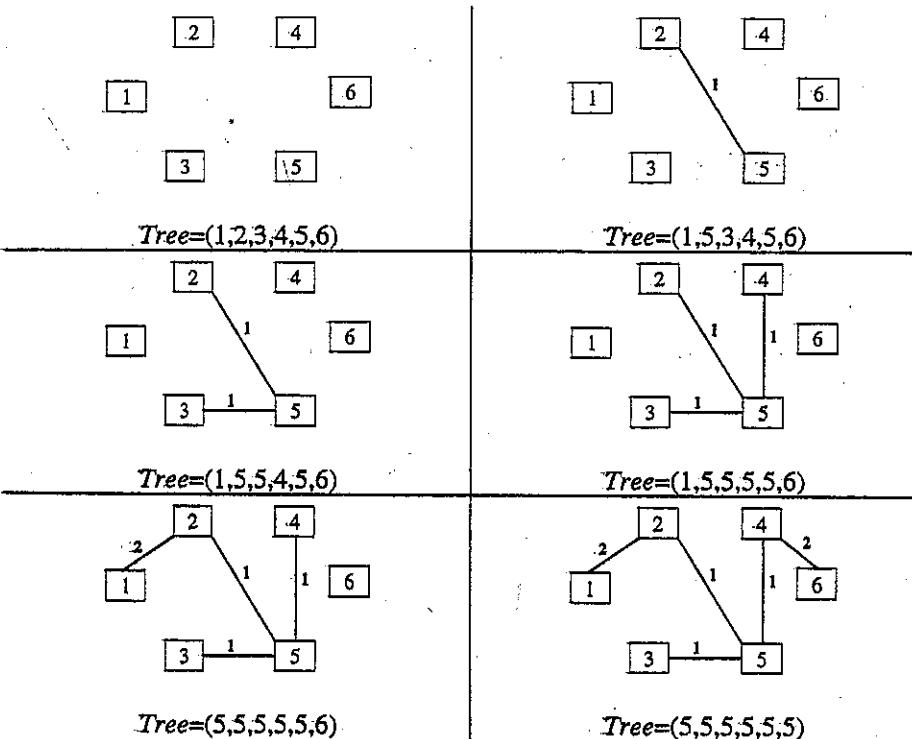
Fie  $G=(V,E)$  un graf neorientat conex, cu costuri asociate muchiilor. Un arbore parțial al lui  $G$  este un graf parțial conex și fără cicluri. Realizați un program care determină un arbore parțial de cost minim, adică un arbore parțial pentru care suma costurilor tuturor muchiilor sale este minimă.

#### Solutie:

Considerăm  $N$  cardinalul mulțimii  $V$  și  $M$  cardinalul mulțimii  $E$ . Algoritmul lui Kruskal construiește arborele parțial de cost minim, pornind de la  $N$  arbori disjuncti, notați  $T_1, T_2 \dots T_N$ . Fiecare vârf al grafului definește la momentul inițial, cîte un arbore. La fiecare pas al algoritmului vor fi aleși doi arbori care se vor unifica. În finalul algoritmului se va obține un singur arbore care va constitui un APM al grafului  $G$ . Proprietatea de acoperire minimă a arborelui rezultat este dată de modul de alegere a celor doi arbori care vor fi unificați la fiecare etapă a algoritmului. Regula respectată este următoarea:

Se identifică muchia de cost minim, nefolosită anterior, și care are vîrfurile extreme în doi arbori disjuncti. În felul acesta, prin unirea celor doi arbori va rezulta tot un arbore (nu se vor crea cicluri).

Să privim modul de construcție al A.P.M. cu ajutorul algoritmului lui Kruskal pe graful luat ca exemplu la prezentarea algoritmului lui Prim: Vectorul  $Tree$  codifică arborii disjuncti astfel:  $Tree[x]=y$  semnifică faptul că vârful  $x$  se află în arborele cu numărul de ordine  $y$ .



S-a obținut un A.P.M plecând de la nodul 1, costul acestuia fiind 7.

În transcrierea algoritmului în pseudocod, mulțimea  $A$  desemnează muchiile arborelui de acoperire minimă. Subprogramul  $reuneste(x,y)$  realizează unificarea arborilor disjuncti în care se regăsesc nodurile  $x$  și  $y$ .

```

1 Kruskal (G=(V,E,cost))           //complexitate O(N*M)
2   A = Ø;                         //lista muchilor din A.P.M
3   pentru orice vîrf i din V execută
4     Tree[i] ← i
5
6   Sortare a listei muchilor ∈ E, crescator după cost
7   pentru fiecare (x,y) ∈ E execută
8     dacă Tree[x]≠Tree[y] atunci
9       A ← { (x,y) }; reuneste(x,y);
10
11
12   returneaza A
13

```

Implementarea în limbaj a algoritmului lui *Kruskal*, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=101;
MAX_M=1001;
INF=maxint div 2;
type muchie=record
  x,y,c:integer;
end;
var n,m,cost:integer; x:muchie;
e:Array[0..max_m]of muchie;
T:Array[0..max_n]of integer;

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_N 101
#define MAX_M 1001
#define INF 0x3f3f
struct muchie
{ int x, y, c; } e[MAX_M];
int N, M, T[MAX_N], cost;

```

Vectorul  $T$  are aceeași semnificație ca și vectorul  $Tree$  prezentat anterior în exemplu și în pseudocod.

```

1 procedure reuneste(i,j:integer);
2 var k:integer;
3 begin
4   i:=T[i]; j:=T[j];
5   if (i<>j) then
6     for k:=1 to n do
7       if T[k]=i then T[k]:=j;
8   end;
9
10 procedure kruskal;
11 var i,j,k,cc:integer;
12 begin
13   for i:=0 to M-1 do
14     for j:=i+1 to M-1 do
15       if e[i].c>e[j].c then begin
16         x:=e[i];
17         e[i]:=e[j];
18         e[j]:=x;
19       end;

```

```

void reuneste(int i, int j)
{
  int k;
  i = T[i];
  j = T[j];
  if (i == j) return;
  for (k = 1; k <= N; k++)
    if (T[k] == i) T[k] = j;

  int comp_muchie(const void *i,
                  const void *j)
  {
    return ((int *) i)[2] -
           ((int *) j)[2];
  }

  void kruskal(void)
  {
    int i, j, k, c;
    qsort(e, M, sizeof(e[0]),
          comp_muchie);
    for (i=1;i<=N;i++) T[i]=i;
  }
}

```

```

20   for i:=1 to n do T[i]:=i;
21   for k:=0 to M-1 do begin
22     i:=e[k].x; j:=e[k].y;
23     cc := e[k].c;
24     if (T[i]==T[j]) then continue;
25     reuneste(i, j);
26     inc(cost,cc);
27     writeln(i," ",j," ",cc);
28   end;
29   writeln('cost minim = ',cost);
30 end;

```

```

for (k = 0; k < M; k++)
{
  i = e[k].x; j = e[k].y;
  c = e[k].c;
  if (T[i]==T[j]) continue;
  reuneste(i, j);
  cost += c;
  printf("%d %d %d\n",i,j,c);
}
printf("Cost minim = %d\n",
      cost);
}

```

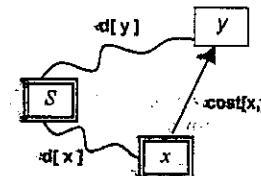
#### 10. Drumuri minime de sursă unică - Algoritmul lui Dijkstra

Fie  $G=(V,E)$  un digraf cu costuri pozitive asociate arcelor. Fiind desemnat un vîrf  $s$  ca punct de plecare - sursă, să se determine pentru orice pereche de vîrfuri  $\{s,x\}$ , costul drumului minim care le unește.

Soluție:

Algoritmul lui *Dijkstra* rezolvă problema enunțată folosind o strategie tip *Greedy*. Notăm cu  $Q$  mulțimea vîrfurilor digrafului și cu  $Use$ , mulțimea vîrfurilor pentru care s-au determinat deja drumurile minime de la sursă. Algoritmul identifică, în cadrul fiecărei iterări, vîrful  $x$  neselectat anterior ( $x \in Q-Use$ ), vîrf ce are proprietatea că este cel mai "apropiat" de sursă. O dată cu identificarea și selectarea lui  $x$ , pentru toate vîrfurile  $y$  adiacente cu el se încearcă o minimizare a costurilor drumurilor prin care acestea sunt legate de sursă. Această operație se efectuează cu ajutorul drumului minim ce trece prin  $x$ .

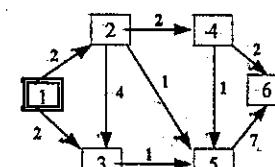
Considerăm că vectorul  $D$  memorează costurile drumurilor minime de la sursă către orice vîrf din graf.



Condiția ca distanța drumului de la  $s$  la  $y$  să poată fi minimizată prin vîrful  $x$  este următoarea:

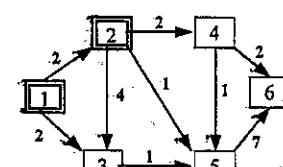
$$d[y] > d[x] + cost[x,y]$$

Să privim modul de operare al algoritmului lui *Dijkstra* pe graful următor, considerând ca sursă nodul 1:



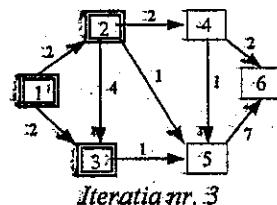
Iterația nr. 1

Vîrful selectat  $x=1$ ,  $D=(0,2,2,\infty,\infty,\infty)$

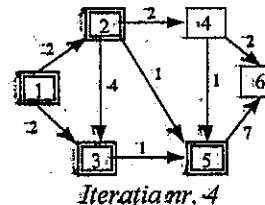


Iterația nr. 2

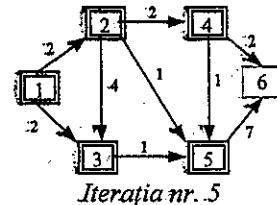
Vîrful selectat  $x=2$ ,  $D=(0,2,2,4,3,\infty)$



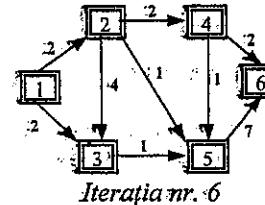
*Iterația nr. 3  
Vârful selectat  $x=3$ ,  $D=(0,2,2,4,3,\infty)$*



*Iterația nr. 4  
Vârful selectat  $x=5$ ,  $D=(0,2,2,4,3,10)$*



*Iterația nr. 5  
Vârful selectat  $x=4$ ,  $D=(0,2,2,4,3,6)$*



*Iterația nr. 6  
Vârful selectat  $x=6$ ,  $D=(0,2,2,4,3,6)$*

Transcrierea în pseudocod a algoritmului folosește următoarele structuri de date:

- tabloul  $D$ , în care elementul  $D[x]$  va reține costul minim de la  $s$  la  $x$ .
- tabloul  $T$ , în care elementul  $T[x]$  va reține vârful precedent lui  $x$  pe drumul minim de la  $s$  la  $x$ .
- Lista  $Q$  conține, pe tot parcursul algoritmului, vârfurile digrafului  $G$  pentru care nu s-au determinat costurile minime ale drumurilor de la sursă către ele.

```

1 Dijkstra (G=(V,E,cost),s)           //complexitate O(N*N)
2   Q ← V;                         //lista Q contine initial toate nodurile din G
3   Use ← ∅;
4   D ← ∞;                         //distantele minime de la s la varfurile din G
5   D[s] ← 0; T[s] = 0;
6   cat timp (Q ≠ ∅) . execută
7     x ← minim(Q);
8     Q ⇒ (x); Use ← (x);
9     pentru fiecare y ∈ Q, adjacent cu x . execută
10    dacă (d[x] + cost[x,y] < d[y]) atunci
11      T[y] = x;
12      D[y] = d[x] + cost[x,y];
13
14
15
16

```

Implementarea în limbaj a algoritmului lui *Dijkstra*, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=101;
const inf=maxint div 2;
var
C:array[0..MAX_N,0..MAX_N]of integer;
D,T:array[0..MAX_N]of integer;
n,m,s,i:integer;
u:Array[0..MAX_N]of byte;
#include <stdio.h>
#include <string.h>
#define MAX_N 101
#define INF 0x3f3f
int N, M, S,i, C[MAX_N][MAX_N],
D[MAX_N], T[MAX_N];
char U[MAX_N];

```

Graful este memorat cu ajutorul matricei costurilor (vezi 2.1.2). Lista  $Q$  a fost codificată cu ajutorul vectorului  $Use$ , astfel  $Use[i]=1$ , dacă pentru vârful  $i$  s-a determinat costul drumului minim și 0, în caz contrar.

Subprogramul *scrie\_drum* poate fi apelat pentru afișarea vârfurilor de pe fiecare drum minim determinat cu ajutorul subprogramului *dijkstra*.

```

1 procedure dijkstra(sursa:integer);
2 var i,min,nod:integer;
3 begin
4   fillchar(u,sizeof(u),0);
5   fillchar(t,sizeof(t),0);
6   fillchar(d,sizeof(d),63);
7   D[sursa]:=0;
8
9   while true do begin
10     min:= INF; nod:=-1;
11     for i:=1 to n do
12       if (u[i]=0)and (min>d[i])
13         then begin
14           min:=D[i];
15           nod:=i;
16         end;
17
18       if (min=inf) then break;
19       U[nod]:=1;
20       for i:=1 to N do
21         if D[i]>D[nod]+C[nod,i]
22         then begin
23           D[i]:=D[nod]+c[nod,i];
24           T[i]:=nod;
25         end;
26       end;
27     end;
28
29 procedure scrie_drum(i:integer);
30 begin
31   if T[i]<>0 then
32     scrie_drum(T[i]);
33   write(i,' ');
34 end;

```

```

void dijkstra(int sursa)
{
  int i, min, nod;
  memset(U, 0, sizeof(U));
  memset(T, 0, sizeof(T));
  memset(D, 0x3f, sizeof(D));
  D[sursa] = 0;

  while (1)
  {
    min = INF;
    nod = -1;
    for (i = 1; i <= N; i++)
      if (!U[i] && min > D[i])
        min = D[i], nod = i;

    if (min == INF) break;
    U[nod] = 1;

    for (i = 1; i <= N; i++)
      if (D[i] > D[nod]+C[nod][i])
      {
        D[i] = D[nod]+C[nod][i];
        T[i] = nod;
      }
  }
}

void scrie_drum(int i)
{
  if (T[i])
    scrie_drum(T[i]);
  printf("%d ", i);
}

```

### 11. Drumuri minime între oricare vârfuri. Algoritmul lui Roy-Floyd

Fie  $G=(V,E)$ , graf orientat cu costuri reale asociate arcelor. Să se determine, pentru orice pereche de vârfuri  $x, y \in V$ , costul drumului minim de la  $x$  la  $y$ .

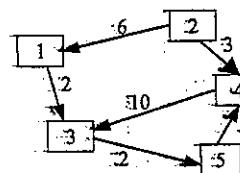
Soluție:

Numim matrice a drumurilor minime, o matrice pătratică  $D(N,N)$  în care valoarea elementului  $D[i,j]$  indică costul minim al unui drum ce leagă vârful  $i$  de vârful  $j$ .

De exemplu, pentru graful ilustrat sălăturat matricea drumurilor minime este următoarea:

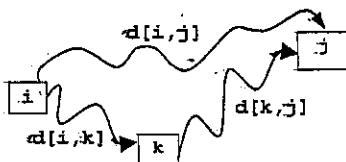
```

0 42 36 4
6 10 18 3 10
10 10 4 2
10 10 0 12
12 2 0
  
```



Algoritmul *Roy-Floyd* construiește matricea drumurilor minime, plecând de la matricea ponderilor - *Cost*.

Înțial, matricea  $D$  reține costurile arcelor prezente în digraf. Algoritmul parcurge de  $n$  ori matricea  $D$ , câte o dată pentru fiecare vârf  $k$  al digrafului. Astfel, între orice pereche de vârfuri  $i$  și  $j$  se încercă minimizarea costului drumului dintre ele prin concatenarea drumurilor de la  $i$  la  $k$  și de la  $k$  la  $j$ .



Regula descrisă mai sus poate fi exprimată astfel:

$$D[i,j] = \min(D[i,j], D[i,k] + D[k,j])$$

Pentru identificarea vârfurilor plasate pe drumurile minime, nu mai este folositoare o informație de tip vârf "predecesor" (tată), deoarece minimizarea se face prin concatenarea a două drumuri și nu a unui drum cu un arc (ca în cazul algoritmului *Dijkstra*). Din această cauză se va construi matricea  $Urm$  care va reține, pentru orice pereche de vârfuri  $i$  și  $j$ , vârful care îi urmează lui  $i$  pe drumul minim către  $j$ .

Transcrierea în pseudocod a algoritmului lui *Roy-Floyd* este următoarea:

```

1 Roy-Floyd (G=(V,E,cost))           /*complexitate: O(N^3)*/
2   D ← Cost;                      //initializare matrice drumuri minime
3   pentru i ← 1..n executa
4     pentru j ← 1..n executa
5       daca (c[i,j]>0) AND (c[i,j]<∞) atunci Urm[i,j] ← j
6         altfel Urm[i,j] ← 0
7
8
9
10  pentru k ← 1..n executa
11    pentru i ← 1..n executa
12      pentru j ← 1..n executa
13        D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
14        Urm[i,j] ← Urm[i,k]
  
```

Implementarea în limbaj sa subprogramului ce realizează construirea matricei drumurilor minime, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=100;
  INF=maxint div 2;
var
  C, D:
  array[0..max_n,0..max_n]of integer;
  Urm:array[0..max_n,0..max_n]of byte;
  m,m:integer;
  
```

Ca și în pseudocod, digraful este considerat să fi memorat cu ajutorul matricei ponderilor (vezi 2.1.2). Complexitatea algoritmului lui *Roy-Floyd* este cubică  $O(N^3)$ .

```

1 procedure floyd;
2 var i,j,k:integer;
3 begin
4   for i:=1 to n do
5     for j:=1 to n do
6       if (C[i,j]>0)AND(C[i,j]<INF)
7         then Urm[i,j]:=j
8         else Urm[i,j]:=0;
9   D:=C;
10  for k:=1 to n do
11    for i:=1 to n do
12      for j:=1 to n do
13        if (D[i,j]>D[i,k]+D[k,j])
14          then begin
15            D[i,j]:=D[i,k]+D[k,j];
16            Urm[i,j]:=Urm[i,k];
17          end;
18      end;
19
20  procedure scrie_drum(i,j:integer);
21 begin
22   write(i, ' ');
23   if i<>j then
24     scrie_drum(Urm[i,j],j)
25   else writeln;
26 end;
  
```

```

void floyd(void)
{
  int i, j, k;
  for (i = 1; i <= N; i++)
    for (j = 1; j <= N; j++)
      if (C[i][j] != 0 && C[i][j] != INF)
        Urm[i][j]=j;
      else Urm[i][j]=0;

  memcpy(D, C, sizeof(C));
  for (k = 1; k <= N; k++)
    for (i = 1; i <= N; i++)
      for (j = 1; j <= N; j++)
        if (D[i][j]>D[i][k]+D[k][j])
        {
          D[i][j]=D[i][k]+D[k][j];
          Urm[i][j]=Urm[i][k];
        }
}

void scrie_drum(int i, int j)
{
  printf("%d ", i);
  if (i!=j)
    scrie_drum(Urm[i][j],j);
  else return;
}
  
```

### 13. Flux maxim într-o rețea de transport - Algoritmul Ford - Fulkerson

Fie o rețea de transport  $G=(V,E)$ , în care pentru fiecare arc există asociată o capacitate și un flux inițial nul. Considerând un vârf  $s$  ca sursă și un vârf  $d$  ca destinație, să se determine fluxul maxim care străbate rețeaua de la sursă la destinație.

### Soluție:

Prezentăm în continuare câteva noțiuni teoretice referitoare la rețelele de transport.

**Definiție.** O rețea de transport este un graf orientat  $G = (V, E)$  care satisfac următoarele condiții:

- există un unic vârf  $s \in V$  în care nu intră nici un arc (grad interior 0) numit sursă;
- există un unic vârf  $d \in V$  din care nu ieșe nici un arc (grad exterior 0) numit destinație;
- pentru fiecare nod  $x \in V - \{s, d\}$  există cel puțin un drum de la sursă la destinație;
- pe mulțimea arcelor se definește o funcție  $C: E \rightarrow \mathbb{R}^+$  numită funcție de capacitate.

**Definiție.** Prin flux în rețeaua de transport  $G$  se înțelege o funcție  $f: E \rightarrow \mathbb{R}^+$  care îndeplinește simultan următoarele condiții :

- Condiția de conservare: pentru orice nod  $x$  din  $V$ , cu excepția sursei și a destinației, suma fluxurilor de pe arcele care intră în  $x$  este egală cu suma fluxurilor de pe arcele care ieș din  $x$ .
- Condiția de mărginire: fluxul de pe orice arc este mai mic sau egal decât capacitatea lui.

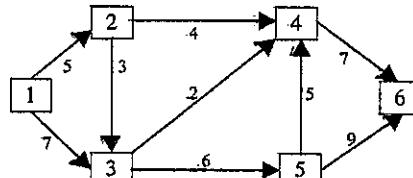
**Definiție.** Numim *graf rezidual* asociat unei rețele de transport un graf  $G_R = (V, E_R)$  cu proprietatea că  $(i, j) \in E_R$ , dacă și numai dacă sunt îndeplinite simultan următoarele condiții:

- $(i, j) \in E$  și b)  $f_{ij} > 0$  sau  $c_{ij} > f_{ij}$ .

Numim *drum de creștere* în graful rezidual un drum de la sursă către destinație.

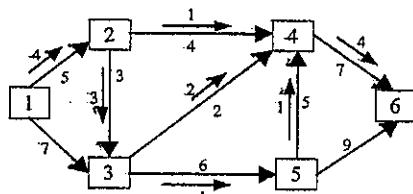
Considerăm rețeaua de transport alăturată. Pentru fiecare arc există asociat un număr reprezentând capacitatea:

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



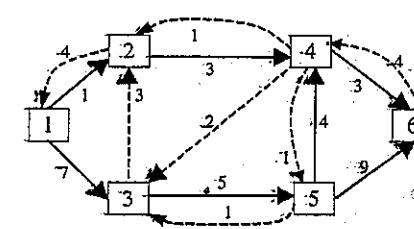
Considerăm că în această rețea de transport există un flux descris în figura alăturată. Pentru fiecare arc există asociat un alt număr reprezentând fluxul introdus:

1 2 5 4	2 3 3 3	4 6 7 4
1 3 7 0	3 4 2 2	5 4 5 1
2 4 4 1	3 5 6 1	5 6 9 0



«Graful rezidual asociat rețelei de transport prezentate este format din următoarele arce:

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



Una dintre cele mai interesante probleme din teoria grafurilor o reprezintă determinarea fluxului maxim într-o rețea, flux care străbate rețeaua de la sursă la destinație. Algoritmul Ford - Fulkerson determină valoarea fluxului maxim și unul dintre acesta.

Pentru rețeaua din exemplul anterior, fluxul maxim este 12, distribuit în felul următor: Arcul (1, 2) - flux 5; Arcul (1, 3) - flux 7; Arcul (2, 3) - flux 1; Arcul (2, 4) - flux 4; Arcul (3, 4) - flux 2; Arcul (3, 5) - flux 6; Arcul (4, 6) - flux 6; Arcul (5, 6) - flux 6

Pașii algoritmului sunt următorii:

repetă  
construiește graful rezidual  $G_R$  asociat rețelei de transport  $G$   
dacă există un drum de creștere atunci saturează acest drum  
până când nu mai există drum de creștere în  $G_R$

Saturarea drumului de creștere se realizează astfel:

- Se alege minimul *min* dintre:
  - $c_{ij} - f_{ij}$ , pentru arcele  $(i, j)$  din drum care corespund unor arce nesaturate din  $G$ ;
  - $f_{ji}$ , pentru arcele  $(i, j)$  care corespund unor arce  $(j, i)$  din  $G$  pe care există flux.
- Pentru fiecare arc  $(i, j)$  din drumul de creștere, arcului asociat din  $G$  i se modifică fluxul atașat astfel:
  - $f_{ij}$  se mărește cu *min* dacă  $c_{ij} > f_{ij}$ ;
  - $f_{ji}$  se micșorează cu *min* dacă  $f_{ji} > 0$ ;

Transcrierea în pseudocod a algoritmului este următoarea:

```

1 Ford_Fulkerson ( G=(V,E,C,Flux),s,d ) //Complexitate: O(N*M^2)
2   Flux ~ 0; //pentru orice arc
3   cat_timp există drum de la s la d în  $G_R$  execută
4     gaseste drum_de_crestere  $s, x_2, \dots, d \in G_R$ ; //  $G_R$  graf rezidual
5
6   m ~ min(  $C_R(x_i, x_{i+1})$  ); //  $C_R$  capacitatea reziduală a arcului
7
8   Flux( $x_i, x_{i+1}$ ) ~ Flux( $x_i, x_{i+1}$ ) + m; // saturează acest drum
9   Flux( $x_{i+1}, x_i$ ) ~ Flux( $x_{i+1}, x_i$ ) - m;
10
11   Flux_Max ~ Flux_Max + m;
12
13 scrie Flux_Max

```

Implementarea în limbaj a subprogramului ce realizează construirea matricei drumurilor minime, prezentată în continuare, nu în considerare următoarele declarații:

```

const MAX_N=101;
    INF=maxint div 2;
var
  C,F:array[0..max_n,0..max_n]of
integer;
  T:array[0..max_n]of integer;
  m,m,s,d,flux:integer;

```

În cadrul surselor prezentate în continuare determinarea drumului de creștere din graful rezidual asociat se realizează printr-o parcurgere BF a acestuia.

Semnificatia variabilelor  $C$ ,  $F$ , Teste următoarea:

- Tabloul  $C$  reține capacitatea tuturor arcelor rețelei de transport;
  - Tabloul  $F$  memorează, pe tot parcursul algoritmului, fluxul de pe fiecare arc al rețelei;
  - Tabloul  $T$  este folosit în parcurgerea  $BF$  pentru reținerea vârfurilor de pe drumul de creștere.

```

1. function min(a,b:integer):integer;
2. begin
3.   if a>b then min:=b
4.     else min:=a;
5. end;
6.
7. function BF(s,d:integer):integer;
8. var p,u,nod,i:integer;
9. Q:array[0..max_n]of integer;
10 begin
11   fillchar(T,sizeof(T),0);
12   Bf:=0;
13   p:=0;
14   u:=0;
15   Q[p]:=s;
16   T[s]:=-1;
17   while p <= u do
18     begin
19       nod := Q[p];
20       for i:=1 to n do
21         if (T[i]=0)and(C[nod,i]>F[nod,i])
22         then
23           begin
24             inc(u);
25             Q[u]:=i;
26             T[i]:=nod;
27             if i = d then bf := 1;
28             end;
29             inc(p);
30           end;
31     end;
32 end;
33
34 int min(int a, int b)
35 {
36   return a < b ? a : b;
37 }
38
39 int BF(int s, int d)
40 {
41   int p, u, nod, i, Q[MAX_N];
42   memset(T, 0, sizeof(T));
43   p = u = 0;
44   Q[p] = s;
45   T[s] = -1;
46   for (; p <= u; p++) {
47     nod = Q[p];
48     for (i = 1; i <= N; i++)
49       if (!T[i] &&
50           C[nod][i] > F[nod][i])
51         { Q[++u] = i;
52           T[i] = nod;
53             if (i == d) return 1;
54         }
55   }
56   return 0;
57 }
58
59 void flux_maxim(void)
60 {
61   int i, r;
62   for (flux=0;BF(S,D);flux += x)
63   {
64     r = INF;
65     i = D;
66   }
67 }

```

```

33 procedure flux_maxim;
34 var i,r:integer;
35 begin
36   flux:=0;
37   while Bf(S,D)<>0 do begin
38     r:=inf;
39     i:=d;
40     while i<>s do begin
41       r:=min(r,C[T[i],i]-F[T[i],i]);
42       i:=T[i];
43     end;
44     i:=d;
45     while i<>s do begin
46       inc(F[T[i],i],r);
47       dec(F[i,T[i]],r);
48       i:=T[i];
49     end;
50     flux:=flux+r;
51   end;
52 end;

```

#### 14. Mouse - Aplicație la algoritmul lui Lee

Pe o tablă dreptunghiulară cu  $n$  linii și  $m$  coloane ( $n, m \leq 100$ ), există zone libere marcate cu 0 și zone cu obstacole marcate cu 1. Știind că pe tablă se află un șoriceł la poziția  $x_i, y_i$  și o bucată de brânză la poziția  $x_b, y_b$ , să se găsească traseul minim pe care trebuie să îl străbată șoricełul până la brânză. Acesta se poate mișca pe tablă în oricare zonă alăturată (pe linie, pe coloană sau pe diagonală) poziției curente, unde nu se află un obstacol.

Din fișierul *mouse.in* se va citi, în ordine, de pe prima linie, numerele  $n, m, x_i, y_i, x_p, y_p$ , iar de pe următoarele  $n$  linii codificarea tablei.

Traseul determinat se va afișa pe ecran, pe o singură linie, în formatul din exemplu.

### *Exemplu.*

*mouse.in*  
463116  
000010  
001100  
001100  
010000

**Sıra afişa**  
**(3,1)(2,2)(1,3)(1,4)(2,5)(1,6)**

*Solutie:*

Pentru rezolvarea problemei să modelăm matricea cu ajutorul unui graf neorientat format din  $n*m$  noduri. Practic, fiecărui nod al grafului îi corespunde un element al matricei și are cel mult opt noduri adiacente.

Problema se rezumă la determinarea celui mai scurt drum între două noduri ale grafului astfel construit. Algoritmul lui Lee rezolvă această cerință folosind strategia parcurgerii în lățime a grafului.

Considerăm că matricea  $a$  memorează inițial codificarea stabiei dreptunghiulare citite după cum urmează:

- $a[x, y] = \infty$  dacă zona  $(x, y)$  este liberă;
- $a[x, y] = -1$  dacă zona  $(x, y)$  este obstacol;
- $a[x, y] = 0$ .

La finalul algoritmului, fiecare element  $a[x, y]$  va reține lungimea lanțului minim dintre nodurile corespunzătoare elementelor  $(x_i, y_i)$  și  $(x, y)$ .

Implementarea acestui procedeu se poate face în felul următor:

- se marchează, în matrice, cu valoarea 1 toți vecinii accesibili nodului de plecare  $x_i, y_i$ . Aceste noduri vor corespunde lanțurilor de lungime 1. Prin vecini accesibili înțelegem vecinii care nu sunt marcați ca obstacole;
- se marchează, în matrice, cu valoarea 2 toți vecinii accesibili nodurilor în care s-a ajuns prin lanțuri de lungime 1, vecini care nu au fost marcați anterior.
- în cadrul iterării  $x$  se vor marca cu  $x+1$  toți vecinii accesibili nodurilor în care s-a ajuns prin lanțuri de lungime  $x$ , vecini care nu au fost marcați anterior.

Algoritmul ia sfârșit când nu se mai poate face nici o nouă marcă sau când s-a marcat elementul  $a[x_f, y_f]$ .

Pentru exemplul prezentat în enunț, la finalul algoritmului, matricea arată după cum urmează:

2	2	2	3	-1	5
1	1	-1	-1	4	5
0	1	-1	-1	4	5
1	-1	2	3	4	5

Implementarea în limbaj a subprogramului, ce codifică algoritmul lui Lee, ia în considerare următoarele declarații:

```
const inf=10000;
dx:array[1..8] of integer=(-1,-1, 0,+1,+1,+1, 0,-1);
dy:array[1..8] of integer=(-0,+1,+1,+1, 0,-1,-1,-1);
var n,m,xi,yi,xf,yf:integer;
a:array[1..100,1..100] of integer;
```

```
#include <stdio.h>
#include <string.h>
#define INF 10000
const int dx[] = {-1,-1, 0,+1,+1,+1, 0,-1};
const int dy[] = { 0,+1,+1,+1, 0,-1,-1,-1};
int N, M, xi, yi, xf, yf;
int A[101][101];
```

Constantele  $dx$  și  $dy$  sunt folosite pentru identificarea vecinilor unui element.

Refacerea traseului minim este realizată în subprogramul  $drum()$  și se bazează pe o traversare a matricei de la  $(x_f, y_f)$  la  $(x_i, y_i)$ . În cadrul subprogramului se identifică elementul vecin din matricea din care s-a ajuns pe lanțul minim în poziția curentă.

Funcția  $verif()$  returnează True (Pascal)/1 (C++) în cazul în care elementul  $a[x, y]$  nu este accesibil din poziția curentă.

```
int verif (int x, int y)
{
    return (x<=0 || y<=0 || x>N || y>M || A[x][y]==-1);
}

void lee(void)
{
    int x, y, xx, yy, i, ok;
    for (ok = 1; ok; )
    {
        ok = 0;
        for (x = 1; x <= N; x++)
            for (y = 1; y <= M; y++)
            {
                if (A[x][y]==INF || A[x][y]==-1) continue;
                for (i = 0; i < 8; i++)
                {
                    xx = x+dx[i];
                    yy = y+dy[i];
                    if (verif(xx,yy)) continue;
                    if (a[xx,yy]>a[x,y]+1)
                        then
                            begin
                                a[xx,yy]:=a[x,y]+1;
                                ok:=true;
                            end;
                        end;
                    end;
                end;
            }
        }
    }

void drum(int x, int y)
{
    int i,xx,yy;
    if (x != xi || y != yi)
        for(i=0;i<8;i++)
    {
        xx=x-dx[i];
        yy=y-dy[i];
        if (A[xx][yy]+1==A[x][y])
        {
            drum(xx,yy);
            break;
        }
    }
    printf("(%d,%d)", x, y);
}
```

O altă modalitate de implementare a algoritmului lui Lee se poate face folosind o structură de tip coadă. Aceasta va conține toate nodurile marcate, în ordinea lungimii lanțurilor minime ce le unește cu punctul de plecare. Practic, implementarea este asemănătoare cu parcurgerea *BFS* a unui graf.

În mod identic cu varianta anterioară de implementare, matricea *a* codifică inițial tabla după cum urmează:

- $a[x,y] = \infty$ , dacă zona  $(x,y)$  este liberă;
- $a[x,y] = -1$ , dacă zona  $(x,y)$  este obstacol;
- $a[x,y] = 0$ .

A două variantă de implementare în limbaj a subprogramului, ce codifică algoritmul lui Lee, ia în considerare următoarele declarații:

```
const inf=10000;
dx:array[1..8] of integer=
(-1,-1, 0,+1,+1,+1, 0,-1);
dy:array[1..8] of integer=
( 0,+1,+1,+1, 0,-1,-1,-1);

var n,m,xi,yi,xf,yf:integer;
ok:boolean;
a:array[1..100,1..100] of integer;
q:array[1..10000,1..2] of byte;
```

Refacerea traseului minim se poate realiza cu ajutorul subprogramului *drum()*, prezentat anterior. În mod identic variantei anterioare, funcția *verif()* returnează True (Pascal) / 1 (C++) în cazul în care elementul  $a[x,y]$  nu este accesibil din poziția curentă.

```
procedure lee;
var x,y,xx,yy,i,st,dr:integer;
begin
  q[1,1]:=xi; q[1,2]:=yi;
  st:=1; dr:=1; ok:=false;
  while (st<=dr) and not ok do
  begin
    x:=q[st,1]; y:=q[st,2];
    for i:=1 to 8 do begin
      xx:=x+dx[i]; yy:=y+dy[i];
      if verif(xx,yy) then
        continue;
      if a[xx,yy]=inf then begin
        a[xx,yy]:=a[x,y]+1;
        inc(dr); q[dr,1]:=xx;
        q[dr,2]:=yy;
        if (xx=xf) and (yy=yf) then
          ok:=true;
      end;
    end;
    inc(st);
  end;
end;
```

```
void lee(void)
{
  int x,ok,y,xx,yy,i,st,dr;
  Q[0][0]:=xi, Q[0][1]:=yi;
  st=0; dr=0; ok=0;
  while (st<=dr && !ok)
  {
    x = Q[st][0], y = Q[st][1];
    for (i = 0; i < 8; i++)
    {
      xx=x+dx[i],yy=y+dy[i];
      if (verif(xx,yy)) continue;
      if (A[xx][yy]==INF)
      {
        A[xx][yy]=A[x][y]+1;
        Q[++dr][0]=xx;
        Q[dr][1]=yy;
        if (xx==xf && yy==yf)
          ok=1;
      }
    }
    st++;
}
```

### 15. Robot - Aplicație la algoritmul lui Lee

Un robot culege piese din cele  $n*m$  compartimente ale unei secții. Harta secției este codificată printr-o matrice de  $n$  linii și  $m$  coloane, fiecare element reprezentând numărul de piese din compartimentul respectiv. Robotul este inițial în primul compartiment, codificat în matrice prin elementul situat pe prima linie și pe prima coloană. Deplasarea robotului se face în oricare din compartimentele alăturate, vecine pe direcțiile nord->sud, west->est. Patronul vrea să știe care este fundamentalul minim pe care îl are robotul. Cu alte cuvinte, pentru fiecare compartiment al secției, el vrea să știe care este numărul *minim* de piese care poate fi culese de robot pe un traseu ce pleacă din punctul inițial.

Realizați un program care rezolvă problema patronului.

Din fișierul *robot.in* se citesc, de pe prima linie, numerele  $n$  și  $m$ , iar de pe următoarele  $n$  linii, câte  $m$  numere reprezentând numărul de piese din fiecare compartiment.

În fișierul *robot.out* se va afișa o matrice de  $n$  linii și  $m$  coloane, în care fiecare element reprezintă numărul minim de piese culese de robot pe un traseu ce unește punctul de start cu compartimentul respectiv.

*Exemplu:* Pentru fișierul:

*robot.in*

```
3 4
3 2 1 3
3 0 9 8
9 0 2 1
```

*robot.out*

```
3 5 6 9
6 5 14 16
14 5 7 8
```

*Soluție:*

Problema poate fi rezolvată prin algoritmul lui Lee. Plecând de la matricea inițială (*a*), se va construi o nouă matrice (*b*) care va reprezenta practic soluția problemei noastre.

Considerăm un element oarecare situat pe linia  $x$  și coloana  $y$ . La fiecare iterație a algoritmului, se încearcă o minimizare a valorii elementelor vecine lui  $b[x,y]$  printr-un traseu care trece prin compartimentul  $(x,y)$ .

Să luăm ca exemplu elementul vecin la nord cu  $b[x,y]$ . Dacă  $b[x-1,y] > b[x,y] + a[x-1,y]$ , atunci  $b[x-1,y] = b[x,y] + a[x-1,y]$ . Evident, toate elementele vecine cu  $b[x,y]$  vor fi minimizate, dacă este posibil.

Pe de altă parte, în cadrul unei iterării, toate elementele matricei vor fi parcuse și va fi executată operația de minimizare a valorilor vecinilor lor.

Algoritmul se oprește când matricea *b* se stabilizează, adică în cadrul unei iterării nu s-a reușit efectuarea nici unei iterării.

Implementarea în limbaj a subprogramului ce realizează construirea matricei *b*, prezentată în continuare, ia în considerare următoarele declarații:

```

const inf=maxint div 2;
dx:array[1..4]of integer=
(0,0,1,-1);
dy:array[1..4]of integer=
(1,-1,0,0);
var
a,b:array[0..100,0..100]of integer;
n,m,i,j,y,x,k:integer;
ok:boolean;

```

Înaintea executării subprogramului *lee*, matricea *b* se inițializează cu valoarea *inf*. Constantele *dx* și *dy* sunt folosite pentru identificarea vecinilor unui element.

Traseele minime care au determinat valorile finale din matricea *b* pot fi reconstruite pe baza elementelor din matricea *a*. Lăsăm, ca exercițiu, realizarea acestui subprogram.

```

1 procedure lee;
2 begin
3   b[1,1]:=a[1,1];
4   ok:=true;
5   while ok do begin
6     ok:=false;
7     for x:=1 to n do
8       for y:=1 to m do
9         for i:=1 to 4 do
10        if b[x+dx[i],y+dy[i]]>
11          b[x,y]+a[x+dx[i],y+dy[i]]
12        then begin
13          ok:=true;
14          b[x+dx[i],y+dy[i]]:=
15          b[x,y]+a[x+dx[i],y+dy[i]];
16        end;
17      end;
18    end;

```

```

void lee() {
  b[1][1]=a[1][1];
  ok=1;
  while (ok) {
    ok=0;
    for (x=1;x<=n;x++)
      for (y=1;y<=m;y++)
        for (i=0;i<4;i++)
          if (b[x+dx[i]][y+dy[i]]>
              b[x][y]+a[x+dx[i]][y+dy[i]])) {
            ok=1;
            b[x+dx[i]][y+dy[i]]=
            b[x][y]+a[x+dx[i]][y+dy[i]];
          }
    }
}

```

### 2.2.3 Probleme propuse

1. Realizați o funcție care verifică dacă un șir de *k* (*k* ≤ 25) noduri reprezintă un lanț într-un graf neorientat, memorat cu ajutorul matricei de adiacență. Graful conține *n* noduri (*n* ≤ 25). Șirul, matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

2. Realizați o funcție care verifică dacă un șir de *k* (*k* ≤ 25) noduri reprezintă un lanț elementar într-un graf neorientat, memorat cu ajutorul matricei de adiacență. Graful conține *n* noduri (*n* ≤ 25). Șirul, matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

3. Realizați o funcție care verifică dacă un șir de *k* (*k* ≤ 25) noduri reprezintă un lanț simplu într-un graf neorientat, memorat cu ajutorul matricei de adiacență. Graful conține *n* noduri (*n* ≤ 25). Șirul, matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

4. Realizați o funcție care verifică dacă un graf neorientat, memorat cu ajutorul matricei de adiacență, este graf complet. Graful conține *n* noduri (*n* ≤ 25). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

5. Realizați o funcție care verifică dacă un graf neorientat, memorat cu ajutorul matricei de adiacență, este graf regulat. Graful conține *n* noduri (*n* ≤ 25). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

6. Scrieți un subprogram care pentru un graf neorientat, memorat cu ajutorul matricei de adiacență, afișează nodurile cu gradele maxime. Graful conține *n* noduri (*n* ≤ 25). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor.

7. Realizați o funcție care verifică dacă o matrice pătratică de ordin *n* poate reprezenta matricea de adiacență a unui graf neorientat de *n* noduri. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

8. Realizați o funcție care verifică dacă un graf neorientat, memorat cu ajutorul matricei de adiacență, este graf eulerian. Graful conține *n* noduri (*n* ≤ 25). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

9. Scrieți un subprogram care pe baza matricei de adiacență a unui graf neorientat, construiește și returnează, prin intermediul unui parametru, listele de adiacență asociate grafului. Graful conține *n* noduri (*n* ≤ 25). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul a doi parametri.

10. Scrieți un subprogram care elimină arcul *(x,y)* din liste de adiacență ale unui graf orientat. Digraful conține *n* vârfuri (*n* ≤ 25). Listele de adiacență, numărul de vârfuri din digraf și vârfurile *x, y*, vor fi primite de subprogram prin intermediul parametrilor.

11. Scrieți un subprogram care afișează numărul minim de muchii ce trebuie adăugate pentru ca un graf neorientat cu  $n$  ( $n \leq 25$ ) noduri, memorat cu ajutorul matricei de adiacență, să devină conex. Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul a doi parametri. Afiați și care muchii au fost alese pentru a fi introduse în graf.

12. Scrieți un subprogram care afișează ce muchie poate fi introdusă într-un graf neorientat cu  $n$  ( $n \leq 25$ ) noduri, memorat cu ajutorul listelor de adiacență, pentru ca acesta să conțină o componentă conexă cu număr maxim de noduri. Listele de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul a doi parametri.

13. Realizați un subprogram care elimină dintr-un digraf cu  $n$  ( $n \leq 25$ ) vârfuri, toate arcele corespunzătoare vârfurilor cu gradul exterior maxim. Digaful este memorat cu ajutorul listelor de adiacență. Acestea împreună cu numărul de vârfuri din graf vor fi primite de subprogram prin intermediul a doi parametri.

14. Realizați un subprogram care verifică dacă un graf neorientat, memorat cu ajutorul matricei de adiacență este graf aciclic (nu conține cicluri). Graful conține  $n$  noduri ( $n \leq 25$ ). Rezultatul va fi returnat de subprogram prin intermediul parametrului întreg *ok*. Aceasta va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

15. Scrieți un subprogram care, primind un sir de  $n$  ( $n \leq 25$ ) valori naturale  $\deg_1, \deg_2, \dots, \deg_n$ , construiește un arbore (graf conex fără cicluri) cu  $n$  noduri ale căror grade sunt  $\deg_1, \deg_2, \dots, \deg_n$ . Se asigură că  $\deg_1 + \deg_2 + \dots + \deg_n = 2(n-1)$ . Subprogramul va memora arborele cu ajutorul matricei de adiacență și va afișa muchiile acestuia.

16. Realizați o funcție care verifică dacă într-un graf neorientat, memorat cu ajutorul matricei de adiacență există un ciclu elementar de lungime 4. Graful conține  $n$  noduri ( $n \leq 25$ ). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

17. Realizați un program care afișează, dacă există, un ciclu elementar de lungime 3, într-un graf neorientat, memorat cu ajutorul matricei de adiacență. Graful conține  $n$  noduri ( $n \leq 25$ ). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor.

18. Realizați un subprogram care direcționează muchiile unui graf neorientat, memorat cu ajutorul matricei de adiacență, graf care conține un singur ciclu. În urma direcționării realizate, toate vârfurile vor avea gradul interior cel mult 1.

Graful conține  $n$  noduri ( $n \leq 25$ ). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Direcționarea se va realiza prin modificarea matricei de adiacență.

19. Realizați un subprogram care identifică un nod al unui graf neorientat conex, nod care poate fi eliminat, împreună cu muchiile incidente lui, fără ca graful să își pierde conexitatea. Graful conține  $n$  noduri ( $n \leq 25$ ) și este memorat cu ajutorul matricei de adiacență. Aceasta și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor.

20. Realizați un subprogram care identifică componentele conexe ale unui graf neorientat, care sunt arbori. Graful conține  $n$  noduri ( $n \leq 25$ ) și este memorat cu ajutorul matricei de adiacență. Aceasta și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Subprogramul va afișa pe ieșirea standard, pe câte o linie, toate nodurile fiecărei componente conexe identificate.

21. Realizați o funcție care verifică dacă există vreun vârf al unui digraf cu gradul interior nul. Graful conține  $n$  vârfuri ( $n \leq 25$ ) și este memorat cu ajutorul listelor de adiacență. Aceasta și numărul de vârfuri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0, în caz contrar.

22. Considerăm un digraf cu  $n$  vârfuri ( $n \leq 25$ ), memorat cu ajutorul listelor de adiacență și un vârf  $x$  din graf. Realizați un subprogram care identifică vârfurile legate de  $x$  prin drumuri de lungime minimă pară. Listele de adiacență, numărul de vârfuri din graf și vârful  $x$  vor fi primite de subprogram prin intermediul parametrilor. Subprogramul va afișa pe ieșirea standard, pe câte o linie, vârfurile fiecărui drum determinat.

23. Prin definiție, un graf este bipartit dacă nodurile sale pot fi împărțite în două submulțimi  $S_1, S_2$ , astfel încât oricare două noduri, aparținând aceleiași submulțimi  $S_i$ , nu sunt adiacente: dacă  $\{x, y\} \in S_i \Rightarrow (x, y) \notin E$ . Realizați o funcție care verifică dacă un graf neorientat  $G=(V, E)$ , memorat cu ajutorul matricei de adiacență este graf bipartit. Graful conține  $n$  noduri ( $n \leq 25$ ). Matricea de adiacență și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Funcția va întoarce valoarea 1 în caz afirmativ și 0 în caz contrar.

24. Realizați un subprogram care identifică cele două submulțimi de noduri în care se împart, prin definiție, nodurile unui graf bipartit. Graful conține  $n$  noduri ( $n \leq 25$ ) și este memorat cu ajutorul matricei de adiacență. Aceasta și numărul de noduri din graf vor fi primite de subprogram prin intermediul parametrilor. Subprogramul va afișa pe ieșirea standard, pe două linii, nodurile fiecărei submulțimi.

25. Considerăm un graf neorientat  $G=(V,E)$ . Prin definiție  $G'=(V,E')$  reprezintă complementul lui  $G$  dacă pentru orice pereche de noduri  $x, y$  neadiacente în  $G$  avem  $(x,y)$  muchie în  $G'$ .

Să se realizeze un subprogram care primind, prin intermediul a doi parametri, matricea de adiacență și numărul de noduri al unui graf, creează și returnează listele de adiacență asociate grafului complement.

26. Realizați un program care afișează toate ciclurile hamiltoniene ale unui graf neorientat conex. Graful conține  $n$  noduri ( $n \leq 25$ ). În cazul în care graful nu este hamiltonian, se va afișa mesajul "imposibil".

27. Într-un oraș există  $n$  centre comerciale unite printr-o rețea stradală unidirecțională. Se cere realizarea unui program care identifică un centru comercial în care se poate ajunge din oricare altul. Din fișierul *graf.in* se vor prelua, de pe prima linie, numărul  $n$  de centre și  $m$  de străzi, iar de pe următoarele  $m$  linii, perechi de numere  $i, j$  având semnificația de stradă de la centrul  $i$  la  $j$ .

*Exemplu:* *graf.in*

```
5 5
1 2
3 2
5 3
5 1
4 1
```

Se va afișa: 2.

28. Se consideră un graf orientat cu  $n$  vârfuri. Să se realizeze un program care determină toate drumurile de lungime 2.

Din fișierul *graf.in* se vor prelua, de pe prima linie, numărul  $n$  de vârfuri, iar de pe următoarele linii, perechi de numere  $i, j$  având semnificația arc de la  $i$  la  $j$ .

În fișierul *graf.out*, pe fiecare linie se vor afișa câte trei numere reprezentând vâfurile de pe fiecare drum identificat.

*Exemplu:* Pentru fișierul:

<i>graf.in</i>	<i>graf.out</i>
5	1 2 3
1 2	1 4 5
1 4	2 3 4
2 3	3 4 5
3 4	
4 5	

29. Se consideră un graf neorientat cu  $n$  vârfuri. Să se realizeze un program care determină numărul maxim de muchii care pot fi eliminate astfel încât numărul de componente conexe ale grafului să nu se modifice.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri, iar de pe următoarele linii perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa numărul maxim identificat.

*Exemplu:* Pentru fișierul:

*graf.in*

```
6
1 2
1 4
2 4
3 6
5 6
```

*graf.out*

```
1
```

30. Se consideră un graf neorientat cu  $n$  vârfuri. Să se realizeze un program care determină toate ciclurile elementare de lungime 3 care conțin un vârf cunoscut.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri și vârful dat, iar de pe următoarele linii, perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se vor afișa, pe fiecare linie, vâfurile de pe fiecare ciclu determinat. Primul și ultimul vârf al fiecarui ciclu va fi cel dat.

*Exemplu:* Pentru fișierul:

*graf.in*

```
4 1
1 2
1 4
2 4
2 3
3 4
3 1
```

*graf.out*

```
1 2 4 1
1 4 3 1
```

31. Se consideră un graf neorientat cu  $n$  vârfuri. Se desemnează un vârf ca nod de plecare. Să se realizeze un program care determină toate vâfurile din graf, legate de nodul de plecare prin lanțuri de lungime minimă dată.

Din fișierul *graf.in* se vor prelua, în ordine, de pe prima linie numărul  $n$  de vârfuri, nodul de plecare și lungimea minimă impusă. De pe următoarele linii se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se vor afișa pe fiecare linie, în ordine, vâfurile de pe fiecare lanț determinat. Ultimul nod de pe fiecare linie fiind cel legat de nodul de plecare, printr-un lanț de lungime minimă impusă.

*Exemplu:* Pentru fișierul:

*graf.in*

```
5 1 2
1 4
2 4
3 4
3 5
2 5
```

*graf.out*

```
1 4 2
1 4 3
```

32. Se consideră un graf neorientat cu  $n$  vârfuri. Să se adauge un număr minim de muchii pentru ca graful să nu conțină vârfuri terminale (cu grad 1).

Din fișierul *graf.in* se vor prelua de pe prima linie, numărul  $n$  de vârfuri, iar de pe următoarele linii, se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa, pe prima linie, numărul minim determinat, iar pe fiecare din liniile următoare, perechi de numere reprezentând muchiile adăugate.

*Exemplu:* Pentru fișierul:

<i>graf.in</i>	<i>graf.out</i>
5	2
1 2	2 3
1 3	4 5
1 4	
1 5	

33. Se consideră un graf neorientat cu  $n$  vârfuri. Să se determine un lanț care trece prin trei vârfuri date.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri și cele trei vârfuri date, iar de pe următoarele linii, se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa succesiunea de vârfuri de pe lanțul determinat.

*Exemplu:* Pentru fișierul:

<i>graf.in</i>	<i>graf.out</i>
6 1 5 6	1 3 5 6
1 2	
1 3	
1 4	
3 5	
4 5	
5 6	

34. Se consideră un graf neorientat conex cu  $n$  vârfuri. Să se determine un lanț de lungime minimă care leagă un nod de grad maxim cu unul de grad minim.

Din fișierul *graf.in* se vor prelua, de pe prima linie, numărul  $n$  de vârfuri, iar de pe următoarele linii se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa succesiunea de vârfuri de pe lanțul determinat.

*Exemplu:* Pentru fișierul:

<i>graf.in</i>	<i>graf.out</i>
6	1 3 5 6
1 2	
1 3	
1 4	
3 5	
4 5	
5 6	

35. Se consideră un graf neorientat conex cu  $n$  vârfuri. Să se determine un lanț hamiltonian, pentru care sirul gradelor vârfurilor nu reprezintă un sir monoton.

Din fișierul *graf.in* se vor prelua, de pe prima linie, numărul  $n$  de vârfuri, iar de pe următoarele linii, se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa succesiunea de vârfuri ale pe lanțul hamiltonian determinat. În situația în care nu există un astfel de lanț, în fișier, se va afișa mesajul **imposibil**.

*Exemplu:* Pentru fișierele:

<i>graf.in</i>	<i>graf.out</i>	<i>graf.in</i>	<i>graf.out</i>
4	4 2 1 3	3	imposibil
1 2		1 2	
1 3		1 3	
2 4		2 3	
4 1			

36. Să se realizeze un program care verifică dacă un graf  $G_2$  reprezintă graf parțial al grafului  $G_1$ . Fișierul *graf1.in* conține pe prima linie numărul  $n$  de vârfuri al grafului  $G_1$ , iar pe următoarele linii, perechi de numere reprezentând muchiile lui. Fișierul *graf1.out* conține, în același format, datele referitoare la graful  $G_2$ .

În fișierul *graf.out* se va scrie rezultatul verificării prin mesajul **DA**, dacă  $G_2$  este graf parțial al lui  $G$  sau **NU**, în caz contrar.

*Exemplu:* Pentru fișierele:

<i>graf1.in</i>	<i>graf2.in</i>	<i>graf.out</i>
4	4	DA
1 2	1 2	
1 3	1 3	
2 4	4 1	
4 1		

37. Prin distanță între două vârfuri ale unui graf neorientat se înțelege cea mai mică lungime a unui lanț care le unește. Ecartamentul unui vîrf reprezintă cea mai mare distanță care îl separă de celelalte vârfuri. Să se realizeze un program care identifică ecartamentul unui nod  $x$  al unui graf neorientat conex.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri și nodul  $x$ , iar de pe următoarele linii, se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa, pe prima linie, ecartamentul nodului  $x$ , iar pe a doua linie, succesiunea de vârfuri de pe lanțul care stabilește acest ecartament.

*Exemplu:* Pentru fișierul:

*graf.in*

6 1

1 2

1 3

1 4

3 5

4 5

5 6

*graf.out*

3

1 4 5 6

38. Prin *diametrul* unui graf se înțelege cel mai mare ecartament al nodurilor sale (vezi pr. 37). Să se realizeze un program care identifică diametrul unui graf neorientat conex.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri, iar de pe următoarele linii se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se va afișa pe prima linie diametrul grafului, iar pe a doua linie, nodurile care au stabilit acest diametru.

*Exemplu:* Pentru fișierul:

*graf.in*

6

1 2

1 3

1 4

3 5

4 5

5 6

*graf.out*

4

2 6

39. Prin mulțime *intern stabilă* se înțelege o submulțime  $S$  de noduri a unui graf neorientat conex, maximală în raport cu proprietatea că, oricare două noduri aparținând ei, sunt neadiacente. Considerând un graf  $G$  să se determine mulțimile intern stabile ale lui.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri, iar de pe următoarele linii se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se vor afișa pe fiecare linie, nodurile fiecărei mulțimi intern stabile.

*Exemplu:* Pentru fișierul:

*graf.in*

9

1 2

1 9

3 4

3 9

6 5

5 9

9 7

7 8

*graf.out*

1 7 5 3

2 8 6 4 9

40. Prin mulțime *extern stabilă* se înțelege o submulțime  $S$  de noduri a unui graf neorientat conex, minimală în raport cu proprietatea că oricare nod, ce nu aparține ei, este adiacent cu un nod din submulțime. Considerând un graf  $G$ , să se determine o mulțime extern stabilă a lui.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri, iar de pe următoarele linii se vor citi perechi de numere  $i, j$  având semnificația de muchie de la  $i$  la  $j$ .

În fișierul *graf.out* se vor afișa pe fiecare linie nodurile mulțimii externe stabilă determinate.

*Exemplu:* Pentru fișierul:

*graf.in*

5

1 2

1 3

2 3

3 4

3 5

4 5

*graf.out*

3 5

41. Se consideră un graf neorientat conex cu  $n$  vârfuri,  $n$  impar. Să se realizeze un program care determină numărul minim de muchii care trebuie adăugate pentru ca graful să devină eulerian.

Din fișierul *graf.in* se vor prelua de pe prima linie numărul  $n$  de vârfuri, iar de pe următoarele linii perechi de numere reprezentând muchiile grafului.

În fișierul text *graf.out* pe prima linie se va afișa numărul minim determinat, iar pe următoarele linii, perechi de două numere reprezentând muchiile adăugate.

*Exemplu:* Pentru fișierul:

*graf.in*

5

1 2

1 3

2 3

3 4

3 5

4 5

2 5

*graf.out*

3

2 4

5 1

1 4

## 2.3. Probleme și algoritmi avansati pe grafuri

### 2.3.1 Probleme rezolvate

#### 1. Puncte de articulație - critice

Un nod dintr-un graf  $G=(V, E)$  neorientat conex este punct de articulație (critic), dacă și numai dacă prin eliminarea lui, împreună cu muchiile incidente acestuia, se pierde proprietatea de conexitate. Realizați un program care determină mulțimea punctelor critice dintr-un graf.

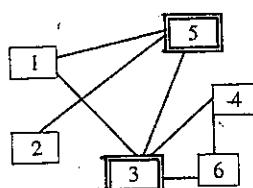
##### Soluție:

Determinarea mulțimii punctelor de articulație poate fi realizată printr-un algoritm liniar  $O(m+n)$ . Acesta are la bază o parcurgere  $DF$  în care se rețin mai multe informații despre fiecare nod, informații care vor conduce, în final, la identificarea punctelor de articulație.

Pentru un nod  $x \in V$  se vor identifica:

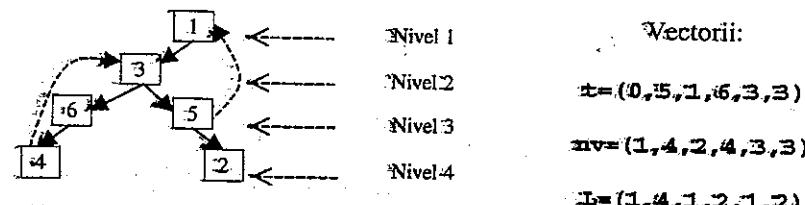
- numărul nivelului atins în parcursarea  $DF$ , memorat în vectorul  $nv$  pe poziția  $x$  ( $nv[x]$ );
- numărul minim al nivelului care poate fi atins din  $x$  folosind descendenții săi și cel mult o muchie de întoarcere. Intuitiv este vorba de "cel mai de sus" nivel care poate fi atins din  $x$  prin intermediul muchiilor de întoarcere accesibile din el sau din descendenții lui. Acest număr va fi reținut în vectorul  $L$  pe poziția  $x$  ( $L[x]$ );
- vârful părinte în arborele  $DF$ , reținut în vectorul  $t$  pe poziția  $x$  ( $t[x]$ ).

Dacă dintr-un nod  $x$  din graf nu se poate ajunge pe un nivel strict mai mic decât al tatălui său din arborele  $DF$  ( $nv[t[x]] \leq L[x]$ ), atunci  $t[x]$  este punct de articulație; eliminarea acestuia, împreună cu muchiile adiacente, ar duce la izolarea nodului  $x$ .



Considerăm graful din figura alăturată. El conține două puncte de articulație: nodul 3 și nodul 5.

Arborele  $DF$  al acestuia cu rădăcina în nodul 1 are patru nivele:



$L[3]=1$  deoarece nivelurile minime atinse de descendenții săi sunt:

- nivelul 2 pentru nodul 4 ( $L[4]=2$ );
- nivelul 1 pentru nodul 5 ( $L[5]=1$ );
- nivelul 2 pentru nodul 6 ( $L[6]=2$ );
- nivelul 4 pentru nodul 2 ( $L[2]=4$ ).

Nivelul minim atins din nodul 3 prin intermediul descendenților săi și al unei muchii de întoarcere este nivelul 1 ( $L[3]=1$ ).

Cum pentru nodul 3 există descendentul direct nodul 6, care nu poate atinge un nivel mai mic decât cel pe care este situat el, rezultă că 3 este punct de articulație. Analog pentru nodul 5.

De reținut că nodul rădăcină al arborelui  $DF$  este punct de articulație dacă are mai mult de un singur descendent direct.

Implementarea în limbaj a subprogramului  $df$  ce identifică mulțimea punctelor critice, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=1001;
type plista^lista;
    lista=record nod:integer;
        urm:Plista;
    end;
var G:array[0..max_n]of plista;
    t,L,nv:array[0..max_n]of integer;
    rad,nr,i,n,m:integer;
    c,u:array[0..max_n]of byte;
```

```
#include <stdio.h>
#define MAX_N 1001
struct lista
{ int nod;
  lista *urm;
 } *G[MAX_N];

int N, M, T[MAX_N], L[MAX_N],
      nv[MAX_N], rad, nr;
char U[MAX_N], c[MAX_N];
```

Vectorul  $C$  va reține pentru fiecare nod, valoarea 0 dacă nodul este critic și 1, în caz contrar. Vectorul  $U$  codifică, în timpul parcurgerii  $DF$ , starea unui nod: vizitat sau nevizitat.

Variabila  $nr$  contorizează numărul de descendenți ai nodului considerat rădăcină în parcursarea  $DF$ .

Graful  $G$  se consideră a fi memorat cu ajutorul listelor de adiacență.

```

1  ...
2  procedure df(nod:integer);
3  var p:plista;
4  begin
5    U[nod]:=1;
6    L[nod]:=nv[nod];
7    p:=G[nod];
8    while p<>nil do begin
9      if (U[p^.nod]=0) then begin
10        nv[p^.nod]:=nv[nod]+1;
11        T[p^.nod]:=nod;
12        if (nod=rad) then inc(nr);
13        DF(p^.nod);
14        if (L[nod]>L[p^.nod]) then
15          L[nod]:=L[p^.nod];
16        if (L[p^.nod]>=nv[nod]) then
17          c[nod]:=1;
18      end
19      else
20        if (p^.nod <>T[nod]) and
21          (L[nod]>nv[p^.nod]) then
22          L[nod]:=nv[p^.nod];
23      p:=p^.urm;
24    end;
25  end;
26
27 begin
28   citeste_graf;
29   for i:=1 to n do
30     if u[i]=0 then begin
31       nv[i]:=1;
32       rad:=i;
33       nr:=0;
34       DF(i);
35       if nr>1 then c[rad]:=1
36       else c[rad]:=0;
37     end;
38   for i:=1 to n do
39     if c[i]<>0 then write(i,' ');
40 end.

```

## 2. Muchii critice – punți

O muchie dintr-un graf  $G=(V, E)$  neorientat conex este puncte (muchie critică), dacă și numai dacă, prin eliminarea sa, se pierde proprietatea de conexitate. Realizați un program care determină mulțimea muchiilor critice dintr-un graf.

### Soluție:

Determinarea mulțimii punțiilor poate fi realizată printr-un algoritm liniar ( $O(m+n)$ ). Acesta are la bază o parcurgere  $DF$  în care se rețin mai multe informații despre fiecare nod, informații care vor conduce, în final, la identificarea punțiilor.

```

... DF(int nod)
{lista *p;
U[nod] = 1;
L[nod] = nv[nod];
for (p = G[nod]; p != NULL;
     p = p->urm)
  if (!U[p->nod]) {
    nv[p->nod] = nv[nod]+1;
    T[p->nod] = nod;
    if (nod == rad) nr++;
    DF(p->nod);
    if (L[nod] > L[p->nod])
      L[nod] = L[p->nod];
    if (L[p->nod] >= nv[nod])
      c[nod] = 1;
  }
else
  if (p->nod != T[nod] &&
      L[nod] > nv[p->nod])
    L[nod] = nv[p->nod];
}

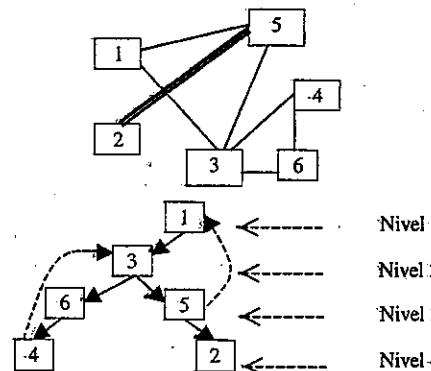
int main(void)
{int i;
citeste_graf();
for (i = 1; i <= N; i++)
  if (!U[i])
    {nv[i] = 1;
    rad = i;
    nr = 0;
    DF(i);
    c[rad] = nr > 1;
    }
  for (i = 1; i <= N; i++)
    if (c[i]) printf("%d ", i);
  return 0;
}

```

Pentru un nod  $x \in V$  se vor identifica:

- numărul nivelului atins în parcurgerea  $DF$ , memorat în vectorul  $nv$  pe poziția  $x$  ( $nv[x]$ );
- numărul minim al nivelului care poate fi atins din  $x$  folosind descendenții săi și cel mult o muchie de întoarcere. Intuitiv este vorba de “cel mai de sus” nivel care poate fi atins din  $x$  prin intermediul muchiilor de întoarcere accesibile din el sau din descendenții lui. Acest număr va fi reținut în vectorul  $L$  pe poziția  $x$  ( $L[x]$ );
- vârful părinte în arborele  $DF$ , reținut în vectorul  $T$  pe poziția  $x$  ( $T[x]$ ).

O observație necesară este că muchiile de întoarcere din arborele  $DF$  nu pot fi muchii critice, deoarece acestea închid un ciclu, iar eliminarea lor nu strică proprietatea de conexitate. Așadar, singurele muchii care vor fi verificate sunt muchiile de arbore. Dacă dintr-un nod  $x$  din graf nu se poate ajunge pe un nivel mai mic sau egal decât al tatălui său din arborele  $DF$  ( $nv[T[x]] < L[x]$ ), atunci muchia ( $t[x], x$ ) este critică.



Considerăm graful din figura alăturată. El conține o muchie critică între nodurile 2 și 5.

Arborele  $DF$  al acestuia cu rădăcina în nodul 1 are patru nivele:

Vectorii:

$$t=(0, 5, 1, 6, 3, 3)$$

$$nv=(1, 4, 2, 4, 3, 3)$$

$$L=(1, 4, 1, 2, 1, 2)$$

Nivelul minim atins din nodul 2, prin intermediul descendenților săi și al unei muchii de întoarcere este nivelul 4 ( $L[2]=4$ ), iar nivelul predecesorului său (nodul 5) este 3 ( $nv[T[2]]=3$ ).

Implementarea în limbaj a subprogramului  $DF$  ce identifică multimea muchiilor critice, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=1001;
type plista=^lista;
 lista=record
  nod:integer;c:boolean;urm:Plista;
 end;
var G:array[0..max_n]of plista;
 t,L,nv:array[0..max_n]of integer;
 i,n,m:integer; p:plista;
 u:array[0..max_n]of byte;

```

```

#include <stdio.h>
#define MAX_N 1001
struct lista {
  int nod; char c;
  lista *urm;
} *G[MAX_N];
int N, M, T[MAX_N], L[MAX_N],
nv[MAX_N];
char U[MAX_N];

```

Vectorul  $U$  codifică, în timpul parcurgerii  $DF$ , starea unui nod: vizitat sau nevizitat.

Graful  $G$  se consideră a fi memorat cu ajutorul listelor de adiacență, iar pentru fiecare element din listele de adiacență se va memora o variabilă de tip boolean(Pascal)/char(C/C++) care va fi *true*/1 dacă muchia respectivă este critică.

```

1  ...
2  procedure df(nod:integer);
3  var p:plista;
4  begin
5    U[nod]:=1;
6    L[nod]:=nv[nod];
7    p:=G[nod];
8    while p<>nil do begin
9      if (U[p^.nod]=0) then begin
10        nv[p^.nod]:= nv[nod]+1;
11        T[p^.nod]:= nod;
12        DF(p^.nod);
13        if (L[nod]>L[p^.nod]) then
14          L[nod]:= L[p^.nod];
15        if (L[p^.nod]>nv[nod])then
16          p^.c := true;
17      end
18      else
19        if (p^.nod <>T[nod])and
20          (L[nod] > nv[p^.nod])then
21            L[nod]:= nv[p^.nod];
22        p:=p^.urm;
23      end;
24    end;
25
26  begin
27    citeste_graf;
28    for i:=1 to n do
29      if u[i]=0 then
30        begin
31          nv[i]:=1;
32          DF(i);
33        end;
34    for i:=1 to n do
35      begin
36        p:=G[i];
37        while p<>nil do
38          begin
39            if p^.c then
40              write(i,' ',p^.nod);
41            p:=p^.urm;
42          end;
43        end;
44      end.
45

```

```

1  ...
2  void DF(int nod)
3  {lista *p;
4  U[nod] = 1;
5  L[nod] = nv[nod];
6  for (p = G[nod]; p != NULL;
7       p = p->urm)
8    if (!U[p->nod]) {
9      nv[p->nod] = nv[nod]+1;
10     T[p->nod] = nod;
11     DF(p->nod);
12     if (L[nod] > L[p->nod])
13       L[nod] = L[p->nod];
14     if (L[p->nod] > nv[nod])
15       p->c = 1;
16   }
17   else
18     if (p->nod != T[nod] &&
19         L[nod] > nv[p->nod])
20       L[nod] = nv[p->nod];
21
22   int main(void)
23   {int i;
24   lista *p;
25   citeste_graf();
26   for (i = 1; i <= N; i++)
27     if (!U[i])
28       { nv[i] = 1;
29         DF(i);
30       }
31   for (i = 1; i <= N; i++)
32     for (p = G[i]; p != NULL;
33          p = p->urm) if (p->c)
34       printf("%d %d\n", i, p->nod);
35   return 0;
36

```

### 3. Componente biconexe

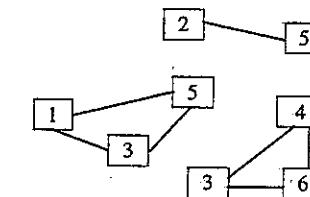
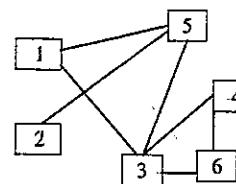
Prin definiție, un graf  $G=(V, E)$  este biconex dacă nu conține puncte de articulație. Prin componentă biconexă se înțelege un subgraf maximal în raport cu proprietatea de biconexitate. Realizați un program care determină muchiile fiecărei componente biconexe a unui graf.

#### Solutie:

Algoritmul de determinare a componentelor biconexe are la bază parcurgerea  $DF$  a grafului, de aici și complexitatea liniară a acestuia ( $O(m+n)$ ). De fapt, algoritmul este o extensie a algoritmului pentru determinarea punctelor de articulație:

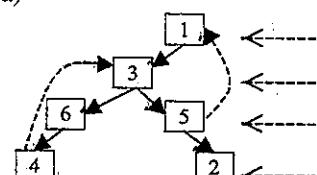
- parcurgerea  $DF$  pentru determinarea numărului minim al nivelului care poate fi atins din fiecare nod folosind descendenții acestuia și cel mult o muchie de întoarcere. Aceste numere vor fi reținute în vectorul  $L$  pe poziția  $x$  ( $L[x]$ ).
- în timpul parcurgerii  $DF$  se vor afișa muchiile din fiecare componentă biconexă. Această operație se va realiza memorându-se explicit o stivă cu muchiile parcurse. Când se determină un nod  $x$  din graf care nu poate ajunge pe un nivel strict mai mic decât al tatălui său din arborele  $DF$  ( $nv[t[x]] \leq L[x]$ ), se va afișa o nouă componentă biconexă. Ea va fi formată din muchiile din stivă, operația de extragere din stivă se oprește la găsirea muchiei  $(t[x], x)$  în vârful stivei.

Considerând ca exemplu graful următor, se vor obține trei componente biconexe:



Etapele algoritmului:

a)



Niv 1  
Niv 2  
Niv 3  
Niv 4

După parcurgerea  $DF$  se vor obține vectorii:  
 $t=(0,5,1,6,3,3)$   
 $nv=(1,4,2,4,3,3)$   
 $L=(1,4,1,2,1,2)$

b)

### Pasul 1:

În momentul găsirii nodului 6 care nu poate ajunge mai sus, stiva conține:  
 $st=((1,3),(3,6),(4,6),(3,4))$ .

Se elimină muchia din stivă până la găsirea muchiei  $t[6].6=(3,6)$ .

La final stiva va conține:

$st=((1,3))$

### Pasul 2:

Se găsește nodul 2 care nu poate ajunge mai sus:

$st=((1,3),(3,5),(2,5))$

Se afișează componenta biconexă formată din succesiunea de muchii până la muchia (2,5).

La final stiva va conține:

$st=((1,3),(3,5))$

Implementarea în limbaj C a subprogramului *DF* care identifică componente biconexe, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=101; MAX_M=1001;
type plista^=lista;
    lista=record
        nod:integer; urm:plista;
    end;
var G:array[0..MAX_N]of Plista;
U,T,L,nv:array[0..MAX_N]of
integer;
st:array[0..MAX_M,0..2]of
integer;
lung,N,M:integer;
```

Vectorul *U* codifică, în timpul parcurgerii *DF*, starea unui nod: vizitat sau nevizitat. Subprogramele *push()* și *pop()* implementează operațiile de introducere, respectiv extragere din stivă a unei muchii.

```
1 ... 
2 procedure push(i,j:integer);
3 begin
4     st[lung]:=i;st[lung+1]:=j;
5     inc(lung);
6 end;
7 procedure pop(var i,j:integer);
8 begin
9     dec(lung); i:=st[lung,0];
10    j:=st[lung][1];
11 end;
```

### Pasul 3:

Se găsește nodul 3 care nu poate ajunge mai sus:

$st=((1,3),(3,5),(1,5))$

Se afișează componenta biconexă formată din succesiunea de muchii până la muchia (1,3).

### Pasul 4:

Stiva vidă, algoritmul ia sfârșit.

### Pasul 5:

Stiva vidă, algoritmul ia sfârșit.

Se afișează componenta biconexă formată din succesiunea de muchii până la muchia (2,5).

La final stiva va conține:

$st=((1,3),(3,5))$

```
#include <stdio.h>
#define MAX_N 101
#define MAX_M 1001
struct lista
{ int nod;
    lista *urm; } *G[MAX_N];
int N, M, T[MAX_N], L[MAX_N],
nv[MAX_N], st[MAX_M][2], lung;
char U[MAX_N];
```

```
12 procedure DF(nod:integer);
13 var p:plista; x,y:integer;
14 begin
15     U[nod]:=1;
16     L[nod]:= nv[nod];
17     p:=G[nod];
18     while p<>nil do begin
19         if (p^.nod<>T[nod]) and
20             (nv[nod]>nv[p^.nod]) then
21             push(p^.nod,nod);
22         if U[p^.nod]=0 then begin
23             nv[p^.nod]:=nv[nod]+1;
24             T[p^.nod]:= nod;
25             DF(p^.nod);
26             if (L[nod]>L[p^.nod]) then
27                 L[nod]:= L[p^.nod];
28             if (L[p^.nod]>=nv[nod])then
29                 begin
30                     repeat
31                         pop(x, y);
32                         write('(',x,',',y,') ');
33                     until not ((x<>nod) or
34                         (y<>p^.nod))
35                         or not ((x<> p^.nod)
36                         or (y<>nod));
37                     writeln;
38                 end;
39             end else
40             if (p^.nod<>T[nod]) and
41                 (L[nod]>nv[p^.nod]) then
42                 L[nod]:= nv[p^.nod];
43                 p:=p^.urm;
44             end;
45         end;
```

```
void DF(int nod)
{
    lista *p;
    int x, y;
    U[nod] = 1; L[nod] = nv[nod];
    for (p = G[nod]; p != NULL;
        p = p->urm)
        if (p->nod != T[nod] &&
            nv[nod] > nv[p->nod])
            push(p->nod, nod);
    if (!U[p->nod])
        {nv[p->nod] = nv[nod]+1;
        T[p->nod] = nod;
        DF(p->nod);
        if (L[nod] > L[p->nod])
            L[nod] = L[p->nod];
        if (L[p->nod] >= nv[nod])
            {do
                {
                    pop(&x, &y);
                    printf("%d %d ", x, y);
                } while ((x != nod ||
                          y != p->nod) &&
                        && (x != p->nod ||
                            y != nod));
            printf("\n");
        }
    else
        if (p->nod != T[nod] &&
            L[nod] > nv[p->nod])
            L[nod] = nv[p->nod];
}
```

### 4. Algoritmul lui Dantzig

Se consideră un graf orientat  $G=(V, E)$  și o funcție cost:  $Ex R+$ . Multimea  $V$  conține  $n$  vârfuri. Se desemnează un vârf  $p$  de plecare. Pentru orice vârf  $i \in V$  se cere determinarea tuturor drumurilor de la  $p$  la  $i$  care au costul minim.

#### Soluție:

Pentru rezolvarea problemei se va folosi algoritmul lui *Dantzig*. Trebuie reținut că acest algoritm obține un graf parțial al lui  $G=(V, E)$  în care orice drum ce pleacă din  $p$  este minim. De fapt, pe parcursul algoritmului se elimină arce ale grafului  $G$ , arce care nu participă la construirea nici unui drum de cost minim cu plecare din  $p$ .

Strategia algoritmului se bazează pe o selectare a vârfurilor în ordinea apropierii lor de nodul  $p$  de plecare. Pe baza costurilor minime determinate succesiv, la fiecare pas al parcurgerii se identifică un nou vârf, cel mai apropiat față de nodul  $p$  de plecare. Notăm cu  $k$  acest nod.

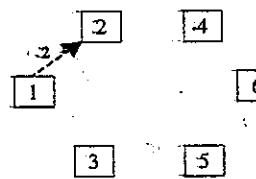
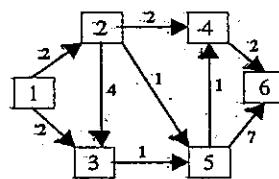
Urmează procesul de identificare a arcelor care se vor adăuga grafului parțial, deoarece conduce la obținerea unor drumuri de la  $p$  la  $k$  cu costul minim determinat. Aceste arce respectă simultan condițiile:

- au extremitatea inițială într-un nod  $i$  deja selectat;
- au extremitatea finală în nodul  $k$ ;
- $d[i] + cost[i, k] = \min$ ; ( $d[i]$  distanța minimă de la  $p$  la  $i$ , iar  $cost[i, k]$ , costul arcului de  $i$  la  $k$ ).

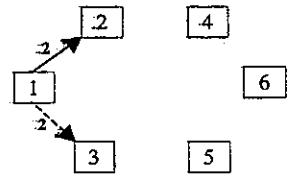
Algoritmul se oprește fie când au fost selectate toate nodurile, fie când nu mai există drumuri de la  $p$  către vârfurile neselectate.

Regăsirea tuturor drumurilor minime de la nodul  $p$  la un nod  $j \in V$  se poate face printr-o căutare recursivă în graful parțial obținut.

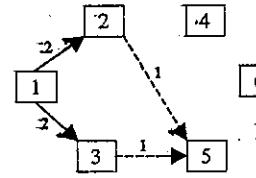
Considerăm ca exemplu graful următor:



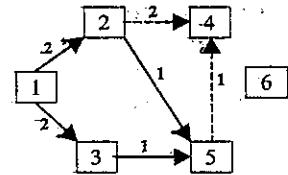
Nodul 2 este selectat;  $d[2]=2$   
Arce adăugat la graful parțial: (1,2).



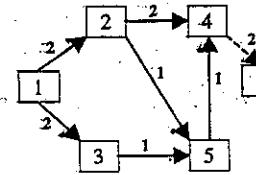
Nodul 3 este selectat;  $d[3]=2$   
Arce adăugate la graful parțial: (1,3).



Nodul 5 este selectat;  $d[5]=3$   
Arce adăugate la graful parțial: (2,5) și (3,5).



Nodul 4 este selectat;  $d[4]=4$   
Arce adăugate la graful parțial: (2,4) și (5,4).



Nodul 6 este selectat;  $d[6]=6$   
Arce adăugate la graful parțial: (4,6).

Pentru exemplul considerat, costul minim de la 1 la 6 este 6 și se obține prin oricare dintre drumurile:

1.2.4.6

1.2.5.4.6

1.3.5.4.6

Implementarea în limbaj a subprogramului *built* care codifică algoritmul lui *Dantzig*, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=101;
const inf=maxint div 2;
var
C,B:array[0..MAX_N,0..MAX_N]of integer;
D,St:array[0..MAX_N]of integer;
min,p,k,j,n,m,s,i:integer;
U:array[0..MAX_N]of byte;
```

```
#include <stdio.h>
#include <string.h>
#define MAX_N 101
#define INF 0x3f3f
int N, M, S,i, C[MAX_N][MAX_N],
B[MAX_N][MAX_N], D[MAX_N],
St[MAX_N];
char U[MAX_N];
```

Graful este memorat cu ajutorul matricei costurilor (vezi 2.1.2). Vectorul  $U$  codifică, pentru fiecare nod, dacă s-a determinat sau nu valoarea drumului minim de la  $p$  către el. Subprogramul *minimum()* determină, la fiecare apel, cel mai apropiat nod neselectat de nodul  $p$  și costul acestuia.

```
procedure minimum;
var i,j:integer;
begin
min:=inf;
for i:=1 to n do
  for j:=1 to n do
    if (U[i]=1)and(U[j]=0) then
      if (min>D[i]+C[i,j]) then
        begin
          min:= D[i]+C[i,j];
          k:=j;
        end;
    end;
end;
procedure build;
var i,j:integer;
begin
for i:=1 to n-1 do begin
  minim;
  if min<>inf then begin
    U[k]:=1;
    D[k]:=min;
    for j:=1 to n do
      if (U[j]=1) and
(D[j]+C[k,j]=min)
        then B[j,k]:=C[j,k];
  end;
end;
end;
```

```
void minim(void)
{
int i,j;
min=inf;
for (i=1;i<=n;i++)
  for (j=1;j<=n;j++)
    if (U[i] && !U[j])
      if (min> D[i]+C[i][j])
        min= D[i]+C[i][j];
        k=j;
}
}

void build(void)
{
int i,j;
for (i=1;i<=n-1;i++) {
  minim();
  if (min!=inf) {
    U[k]=1;
    D[k]=min;
    for (j=1;j<=n;j++)
      if (U[j]&&
(D[j]+C[k][j]==min))
        B[j][k]=C[j][k];
  }
}
}
```

## 5. Algoritmul lui Dijkstra – implementare folosind heap-uri binare

Fie  $G = (V, E)$  un digraf cu costuri pozitive asociate arcelor. Fiind desemnat un vârf  $s$  ca punct de plecare – sursă, să se determine pentru orice pereche de vârfuri  $\{s, x\}$ , costul drumului minim care le unește.

### Solutie:

Algoritmul lui Dijkstra rezolvă problema enunțată folosind o strategie tip Greedy și a mai fost prezentat în capitolul 2.2.2. În continuare vom arăta cum se poate folosi un min-heap pentru a obține complexitate  $O(M * \log N)$ .

Considerând notațiile din capitolul 2.2.2, vom implementa mulțimea  $Q$  folosind un min-heap, astfel extragerea minimului va avea complexitate  $O(\log N)$  (se șterge elementul din vârful heap-ului). Când se modifică distanțele în vectorul  $D$  este necesară și o ridicare în heap a nodului modificat. Pentru a găsi în  $O(1)$ , pe ce poziție se află fiecare nod în heap, se mai folosește un vector suplimentar  $poz[]$  cu aceste informații.

Implementarea în limbaj a algoritmului lui Dijkstra, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=101;
    inf=maxint div 2;
type plista^lista;
    lista=record
        nod,c:integer;urm:Plista;end;
    sir=array[0..max_n]of
integer;
var G:array[0..max_n]of plista;
    D,T,H,poz:sir; n,m,s,nh:integer;
    u:array[0..max_n]of byte;
```

```
#include <stdio.h>
#include <string.h>
#define MAX_N 101
#define INF 0x3f3f
struct lista
{
    int nod, c; lista *urm;
} *G[MAX_N];
int N, M, S, D[MAX_N], T[MAX_N],
H[MAX_N], poz[MAX_N], nh;
char U[MAX_N];
```

Vă puteți reaminti operațiile asociate heap-urilor consultând secțiunea 1.2.3.

```
procedure swap(i,j:integer);
var t:integer;
begin
    t:=H[i]; H[i]:= H[j]; H[j]:=t;
    poz[H[i]]:= i; poz[H[j]]:= j;
end;

procedure HeapDw(r,k:integer);
var St,Dr,i:integer;
begin
    if 2*r+1 <= k then begin
        St := H[2*r+1];
        if 2*r+2 <= k then
            Dr := H[2*r+2]
        else Dr := St-1;
        if St > Dr then i := 2*r+1
        else i := 2*r+2;
    end;
    void swap(int i, int j)
    {
        int t;
        t=H[i]; H[i]=H[j]; H[j]=t;
        poz[H[i]] = i; poz[H[j]] = j;
    }

    void HeapDw(int r, int k)
    {
        int St,Dr,i;
        if (2*r+1<=k)
        {
            St=H[2*r+1];
            if (2*r+2 <=k) Dr=H[2*r+2];
            else Dr=St-1;
            if (St>Dr) i=2*r+1;
            else i=2*r+2;
        }
    }
}
```

```
if D[H[r]]>D[H[i]] then begin
    swap(r,i); HeapDw(i,k)
end;
end;
end;
procedure HeapUp(k:integer);
var t:integer;
begin
    if k > 0 then begin
        t := (k-1) div 2;
        if D[H[k]]<D[H[t]]
        then begin
            swap(k,t); HeapUp(t);
        end;
    end;
end;
procedure BuildH(k:integer);
var i:integer;
begin
    for i:=1 to k-1 do HeapUp(i);
end;
function scoate_heap:integer;
begin
    swap(0, nh-1);
    poz[H[nh-1]]:= 0; dec(nh);
    HeapDw(0, nh-1);
    scoate_heap:=H[nh];
end;
procedure dijkstra(sursa:integer);
var i,nod:integer; p:Plista;
begin
    fillchar(u,sizeof(u),0);
    fillchar(t,sizeof(t),0);
    fillchar(d,sizeof(d),63);
    for i:=0 to N-1 do begin
        poz[i+1]:=i; h[i]:=i+1;
    end;
    D[sursa]:= 0;
    BuildH(n); nh:=n;
    while (nh > 0) do begin
        nod:=scoate_heap; p:=G[nod];
        while p<>nil do begin
            if (D[p^.nod]>D[nod]+p^.c)
            then begin
                D[p^.nod]:= D[nod]+p^.c;
                T[p^.nod]:= nod;
                HeapUp(poz[p^.nod]);
            end;
            p:=p^.urm;
        end;
    end;
end;
void HeapUp(int k)
{
    int t;
    if (k <= 0) return;
    t = (k-1)/2;
    if (D[H[k]]<D[H[t]])
    {swap(k,t);
    HeapUp(t);
    }
}
void BuildH(int k)
{
    int i;
    for (i=1;i<k;i++) HeapUp(i);
}

int scoate_heap()
{
    swap(0, nh-1);
    poz[H[nh-1]]=0; nh--;
    HeapDw(0, nh-1);
    return H[nh];
}

void dijkstra(int sursa)
{
    int i, nod; lista*p;
    memset(U, 0, sizeof(U));
    memset(T, 0, sizeof(T));
    memset(D, 0x3f, sizeof(D));
    for (i = 0; i < N; i++)
    {
        H[i] = i+1; poz[i+1] = i;
    }

    D[sursa]=0; BuildH(N); nh=N;
    while (nh > 0) {
        nod = scoate_heap();
        for (p=G[nod]; p; p=p->urm)
        if (D[p->nod]>D[nod]+p->c)
        D[p->nod] = D[nod]+p->c;
        T[p->nod] = nod;
        HeapUp(poz[p->nod]);
    }
}
```

## 6. Algoritmul lui Bellman-Ford

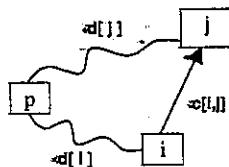
Se consideră un graf orientat  $G=(V, E)$  și o funcție  $c:E \rightarrow R$ . Multimea  $V$  conține  $n$  vârfuri. Se desemnează un vârf  $p$  de plecare. Pentru orice vârf  $j \in V$  se cere determinarea drumului de cost minim de la  $p$  la  $j$ . Se vor detecta situațiile în care există circuite de cost negativ care includ nodul  $p$ .

Soluție:

Algoritmul Bellman-Ford determină drumurile de cost minim dintre un nod desemnat ca sursă (plecare) și restul vâfurilor accesibile lui chiar dacă există costuri negative pe arce. Aceste rezultate sunt furnizate numai în situația în care nodul de plecare nu face parte dintr-un circuit de cost negativ.

Strategia algoritmului este aceea de minimizare succesivă a costului drumului de la  $p$  la orice nod  $j$  din graf ( $D[j]$ ) până la obținerea costului minim.

Această operație se face prin verificarea posibilității ca fiecare arc  $(i,j) \in E$  să participe la minimizarea distanței de la  $p$  la  $j$ . Operația va face o trecere completă prin toate arcele grafului.



Condiția ca distanța de la  $p$  la  $j$  să poată fi minimizată prin arcul  $(i,j)$  este că:

$$d[j] > d[i] + c[i,j].$$

Notăm cu  $n$  numărul de vârfuri ale grafului. Algoritmul efectuează  $n-1$  treceri complete prin multimea arcelor grafului (orice drum elementar poate fi format din maximum  $n-1$  arce).

În final, existența unui circuit negativ face ca la o nouă trecere prin multimea arcelor să fie în continuare posibilă minimizarea costului unui drum. În acest caz algoritmul evidențiază prezența circuitului negativ ce cuprinde nodul sursă.

În situația în care graful este memorat în liste de adiacență, complexitatea algoritmului este  $O(N^2M)$ .

Implementarea în limbaj a subprogramului *bellman*, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=101;
MAX_M=10001;
INF=maxint div 2;
var
D,T:array[0..MAX_N] of integer;
E:array[0..MAX_M,0..3] of integer;
N,M,S:integer;
D[MAX_N], T[MAX_N];
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_N 101
#define MAX_M 10001
#define INF 0x3f3f
```

Graful este memorat cu ajutorul listelor arcelor codificate în tabeloul  $E$  (vezi 2.1.2). Aceasta reține, pentru fiecare arc, vârful initial, vârful final și costul lui.

Subprogramul *drum()* poate fi apelat pentru afișarea vâfurilor de pe fiecare drum minim determinat cu ajutorul subprogramului *bellman*.

```
1 procedure bellman(sursa:integer);
2 var i, j, k, c, ok,nr:integer;
3 begin
4 fillchar(T,sizeof(T),0);
5 fillchar(D,sizeof(D),63);
6 D[sursa]:=0;
7 ok:=1;
8 nr:=1;
9 while (ok<>0)and(nr< N) do
10 {maxim N-1 iteratii}
11 begin
12 ok:=0;
13 k:=0;
14 while k<M do
15 begin
16 i:=e[k,0];
17 j:=e[k,1];
18 c:=e[k,2];
19 if (D[j]>D[i]+c) then
20 begin
21 D[j]:=D[i]+c;
22 T[j]:=i;
23 ok:=1;
24 end;
25 k:=k+1;
26 end;
27 nr:=nr+1;
28 end;
29 for k:=0 to M-1 do
30 begin
31 i:=e[k,0];
32 j:=e[k,1];
33 c:= e[k,2];
34 if (D[j]>D[i]+c) then
35 begin
36 writeln('Ciclu negativ!');
37 halt(0);
38 end;
39 end;
40 end;
41 procedure drum(i:integer);
42 begin
43 if (T[i]<>0) then drum(T[i]);
44 writeln(i);
45 end;
```

```
void bellman(int sursa)
{
int i, j, k, c, ok, nr;
memset(T, 0, sizeof(T));
memset(D, 0x3f, sizeof(D));
D[sursa] = 0;
for (ok = nr = 1; ok && nr < N; nr++)
// maxim N-1 iteratii
for (ok = k = 0; k < M; k++)
{
i = e[k][0];
j = e[k][1];
c = e[k][2];
if (D[j] > D[i]+c)
D[j] = D[i]+c, T[j] = i,
ok = 1;
}

for (k = 0; k < M; k++)
{
i = e[k][0];
j = e[k][1];
c = e[k][2];
if (D[j] > D[i]+c)
{
printf("Ciclu negativ!\n");
exit(0);
}
}

void drum(int i)
{
if (T[i])
drum(T[i]);
printf("%d ", i);
}
```

## 7. Algoritmul lui Bellman-Kalaba

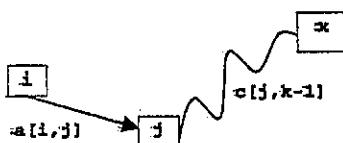
Se consideră un graf orientat  $G=(V, E)$ , conex, fără circuite și o funcție de cost  $a: E \rightarrow R$ . Desemnându-se un vârf  $x$ , se cere determinarea drumurilor de cost maxim de la toate celelalte vârfuri la el, drumuri care includ un număr impus de arce (pași).

### Soluție:

Pe baza matricei costurilor grafului considerat, algoritmul construiește o nouă matrice  $C$  în care, pe coloana  $j$  ( $j \leq n$ ) se vor afla costurile maxime ale drumurilor ce ajung în  $x$  folosind  $j$  arce. Astfel,  $c[i,j]$  reprezintă costul maxim al unui drum de la  $i$  la  $x$  și trece prin  $j$  arce.

Initial, matricea  $C$  va conține, pe prima coloană, costul arcelor spre nodul  $x$ , deci drumuri de lungime 1. A doua iterare va determina costurile maxime ale drumurilor ce ajung în  $x$  și conțin două arce. Ea se va determina pe baza valorilor din coloana anterioară.

Astfel, pentru determinarea valorii  $c[i,2]$  vom adăuga, pe rând, arcele ce pleacă din  $i$  către drumurile de lungime 1, anterior determinate. Se va identifica valoarea maximă dintre acestea:  $c[i,2] = \max\{c[j,1] + a[i,j], j \in V\}$ .



Generalizând, pentru a determina costul maxim al drumului de la  $i$  la  $x$  folosind  $k$  arce se calculează:

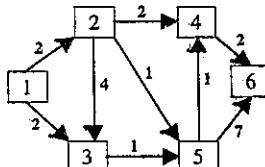
$$\max\{c[j,k-1] + a[i,j], j \in V\}.$$

Numerele de pe fiecare linie a matricei  $C$  vor forma un sir monoton.

Algoritmul se încheie în momentul în care valorile situate pe două coloane consecutive sunt identice, deci când nici un drum nu se mai poate maximiza. Faptul că nu există circuite în graf ne asigură că numărul maxim de iterări ar putea fi egal cu  $n-1$ . Complexitatea algoritmului este  $O(n^3)$ .

Considerăm graful din figură și determinăm drumurile maxime ce ajung în  $x=6$ . Notăm cu  $C^{(i)}$  coloana  $i$  din matricea  $C$  a drumurilor maxime ce ajung în  $x=6$  folosind  $i$  arce.

Algoritmul se va încheia la iterată 5, deoarece  $c^{(4)} = c^{(5)}$ .



$C^{(1)}$	$C^{(2)}$	$C^{(3)}$	$C^{(4)}$	$C^{(5)}$
-1	-1	10	14	14
-1	8	12	12	12
-1	8	8	8	8
2	2	2	2	2
7	7	7	7	7
0	0	0	0	0

Toată strategia algoritmului se bazează pe programarea dinamică.

Implementarea în limbaj a subprogramului *kalaba*, prezentată în continuare, ia în considerare următoarele declarații:

```
const
    MAX_N=100;
var
    a,c,t:array[0..max_n,0..max_n]of
        integer;
    pasi, pl, k, m, n, x:integer;
```

```
#include <stdio.h>
#include <string.h>
#define MAX_N 100
int x, k, pl, pasi, m, n,
    a[MAX_N][MAX_N], c[MAX_N][MAX_N],
    t[MAX_N][MAX_N];
```

Graful este memorat cu ajutorul matricei ponderilor (vezi 2.1.2).

Subprogramul *drum()* poate fi apelat pentru afișarea vârfurilor de pe fiecare drum determinat cu ajutorul subprogramului *kalaba*.

```
1 procedure initializari;
2 var i,j:integer;
3 begin
4     for i:=0 to n do
5         for j:=0 to n do begin
6             c[i,j]:=-1; t[i,j]:=0;
7         end;
8     for j:=1 to n do begin
9         c[j,1]:=a[j,x];
10        if c[j,1]<>-1 then t[j,1]:=j;
11    end;
12 end;
13
14 procedure kalaba;
15 var i,j:integer; ok:boolean;
16 begin
17     k:=2;
18 repeat
19     for i:=1 to n do
20         for j:=1 to n do
21             if (a[i,j]<>-1)and(c[j,k-1]<>-1)
22                 then
23                     if c[i,k]< c[j,k-1]+a[i,j]
24                         then begin
25                             c[i,k]:=c[j,k-1]+a[i,j];
26                             t[i,k]:=j;
27                         end;
28     ok:=true;
29     for i:=1 to n do
30         if c[i,k-1]<>c[i,k] then
31             ok:=false;
32     inc(k);
33     until ok;
34 end;
35
36 procedure drum(x,y:integer);
37 begin
38     write(' ',x);
39     if y>>1 then drum(t[x,y],y-1);
40 end;
41
```

```
void initializari(void)
{
    int i,j;
    for (i=0;i<=n;i++)
        for (j=0;j<=n;j++)
            c[i][j]=-1;
            t[i][j]=0;
    }
    for (j=1;j<=n;j++)
        c[j][1]=a[j][x];
        if (c[j][1]!=-1) t[j][1]=j;
    }
}

void kalaba(void)
{
    int i,j,ok;
    k=2;
    do
    {
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if (a[i][j]!=-1 && c[j][k-1]==-1)
                    if (c[i][k]< c[j][k-1]+a[i][j])
                    {
                        c[i][k]=c[j][k-1]+a[i][j];
                        t[i][k]=j;
                    }
        ok=1;
        for (i=1;i<=n;i++)
            if (c[i][k-1]!=c[i][k])
                ok=0;
        k++;
    }
    while (!ok);
}
```

```
void drum(int x,int y)
{
    cout<<" "<<x;
    if (y!=1) drum(t[x][y],y-1);
}
```

### 8. Algoritmul lui Kruskal (A.P.M.) – implementare folosind păduri de mulțimi disjuncte

Fie  $G=(V,E)$  un graf neorientat conex, cu costuri asociate muchiilor. Un arbore parțial al lui  $G$  este un graf parțial conex și fără cicluri. Realizați un program care determină un arbore parțial de cost minim, adică un arbore parțial pentru care suma costurilor tuturor muchiilor sale este minimă.

#### Soluție:

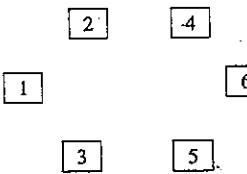
Algoritmul lui *Kruskal* pentru determinarea arborelui parțial de cost minim a mai fost prezentat în capitolul 2.2.2. În continuare vom arăta cum se poate implementa acesta într-o complexitate  $O(M \log^* N)$  folosind o pădure de arbori pentru a menține mulțimile care se construiesc pe parcurs.

Fiecare nod reprezintă un element dintr-o mulțime, și fiecare arbore reprezintă o mulțime. Fiecare element indică doar spre părintele lui, iar rădăcina fiecărui arbore conține reprezentantul mulțimii (care este propriul său părinte).

Pentru a uni două mulțimi se dorește ca rădăcina arborelui cu mai puține noduri să indice spre rădăcina arborelui cu mai multe noduri. Pentru fiecare nod se va menține un "rang", care aproximează logaritmul dimensiunii subarborelui și care este, de asemenea, o margine superioară a înălțimii nodului. Așadar, rădăcina cu rangul cel mai mic va indica spre rădăcina cu rang mai mare.

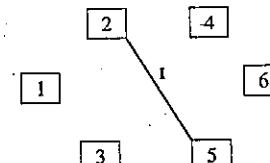
Pentru a verifica dacă două elemente fac parte din aceeași mulțime se determină, pentru fiecare, rădăcina arborelui din care fac parte, și se verifică dacă acestea coincid. O implementare directă a operațiilor descrise ar avea complexitatea  $O(\log N)$  pentru fiecare operație. Folosind o heuristică de "comprimare a drumului" se poate reduce complexitatea fiecărei operații la  $O(\log^* N)$ . După ce se determină reprezentantul unui element (rădăcina arborelui din care face parte) se modifică fiecare nod parcurs în drumul către rădăcina astfel încât să aibă ca părinte rădăcina determinată. Comprimarea drumului nu modifică nici un rang.

Aplicarea acestei structuri de date în algoritmul lui *Kruskal* este evidentă: inițial fiecare nod face parte dintr-o mulțime distinctă, iar introducerea unei noi muchii în arborele de cost minim se face prin reunirea a doi arbori. Să privim modul de construcție al A.P.M. cu ajutorul algoritmului lui *Kruskal* pe graful luat ca exemplu în prezentarea anterioară a acestui algoritm (vectorul  $T$  va reprezenta părintele fiecărui nod).



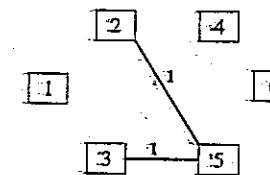
$$T=(1,2,3,4,5,6)$$

$$rang=(0,0,0,0,0,0)$$



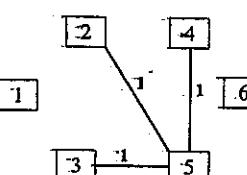
$$T=(1,2,3,4,2,6)$$

$$rang=(0,1,0,0,0,0)$$



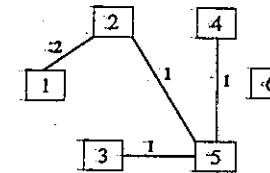
$$T=(1,2,2,4,2,6)$$

$$rang=(0,1;0,0,0,0)$$



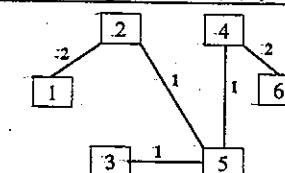
$$T=(1,2,2,2,2,6)$$

$$rang=(0,1,0,0,0,0)$$



$$T=(2,2,2,2,2,6)$$

$$rang=(0,1,0,0,0,0)$$



$$T=(2,2,2,2,2,2)$$

$$rang=(0,1,0,0,0,0)$$

Implementarea în limbaj a algoritmului lui *Kruskal* folosind păduri de mulțimi disjuncte, prezentată în continuare, ia în considerare următoarele declarații:

```
const MAX_N=101;
      MAX_M=1001;
      INF=maxint div 2;
var muchie:array[0..max_m,0..2]of integer;
    T,rang:array[0..max_n]of integer;
    n,m,cost:integer;
```

```
#define MAX_N 101
#define MAX_M 1001
#define INF 0x3f3f

int N, M, muchie[MAX_M][3],
T[MAX_N], rang[MAX_N], cost;
```

Subprogramul *reuneste()* unifică doi arbori conform algoritmului prezentat:

```
1 function multime(n:integer):
2   integer;
3 begin
4   if T[n]<>n then
5     T[n]:=multime(T[n]);
6   multime:=T[n];
7 end;
8 procedure reuneste(i,j:integer):
9 begin
10   i:=multime(i);
11   j:=multime(j);
12   if i<>j then begin
13     if rang[i]<rang[j] then
14       T[i]:=j;
15     else T[j]:=i;
16     if rang[i]=rang[j] then
17       inc(rang[i]);
18   end;
19 end;
```

```
int multime(int n)
{
  if (T[n] != n)
    T[n] = multime(T[n]);
  return T[n];
}

void reuneste(int i, int j)
{
  i=multime(i);
  j=multime(j);
  if (i == j) return;
  if (rang[i] < rang[j])
    T[i] = j;
  else
    T[j] = i;
  if (rang[i] == rang[j])
    rang[i]++;
}
```

```

20 procedure qsort(...);
21 begin
22   ...
23 end;
24
25 procedure kruskal;
26 var i,j,k,cc:integer;
27 begin
28   qsort(muchie,0,M-1);
29   for i:=1 to n do begin
30     T[i]:=i;
31     xang[i]:=0;
32   end;
33   for k:=0 to M-1 do begin
34     i:=muchie[k][0];
35     j:=muchie[k][1];
36     cc:=muchie[k][2];
37     if (multime(i)=multime(j))
38       then continue;
39     reuneste(i, j);
40     inc(cost,cc);
41     writeln(i,' ',j,' ',cc);
42   end;
43 end;

```

```

int comp_muchie(const void *i,
const void *j)
{return ((int*)i)[2] -
((int*)j)[2];
}

void kruskal()
{int i, j, k, x;
qsort(muchie, M,
sizeof(muchie[0]),comp_muchie);

for (i = 1; i <= N; i++) {
T[i] = i; xang[i] = 0;
}
for (k = 0; k < M; k++) {
i = muchie[k][0];
j = muchie[k][1];
c = muchie[k][2];
if (multime(i)==multime(j))
continue;
reuneste(i, j);
cost += c;
printf("%d %d %d\n", i, j, c);
}
}

```

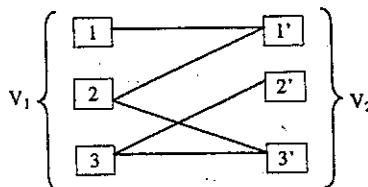
### 9. Cuplaj maxim în graf bipartit

Fie  $G=(V_1, V_2, E)$  un graf neorientat bipartit. Un cuplaj este o submulțime de muchii astfel încât pentru toate vârfurile din  $V_1$  există cel mult o muchie din cuplaj incidentă vârfului. Realizați un program care determină un cuplaj de cardinalitate maximă.

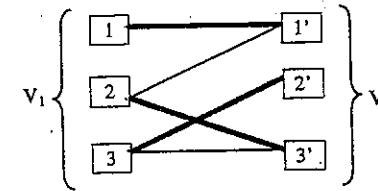
*Exemplu:* Pentru  $\text{card}(V_1)=\text{card}(V_2)=3$ ,  $m=5$  și muchiile  $(1,1')$   $(2,1')$   $(2,3')$   $(3,2')$   $(3,3')$  se va afișa un cuplaj maxim format din 3 muchii:  $(1,1')$   $(2,3')$   $(3,2')$ .

*Soluție:*

Un graf este bipartit dacă nodurile sale pot fi împărțite în două submulțimi  $V_1$ ,  $V_2$ , astfel încât oricare două noduri aparținând aceleiași submulțimi  $V_i$ , nu sunt adiacente: dacă  $(x, y) \in V_i \Rightarrow (x, y) \notin E$ . Vom considera în continuare că  $V_1$  conține  $N_1$  elemente, iar  $V_2$  conține  $N_2$  elemente. Problema găsirii unui cuplaj maxim într-un graf bipartit are numeroase aplicații practice. Să privim graful din exemplu și cuplajul maxim determinat:



Graful inițial



Cuplajul maxim obținut

Se poate obține un algoritm de rezolvare construind o rețea de transport și aplicând algoritmul lui Ford-Fulkerson de flux maxim, totuși implementarea fiind destul de dificilă. Vom prezenta în continuare un algoritm mult mai ușor de implementat, cu complexitate  $O(N_1^2 \cdot N_2)$  (se poate reduce complexitatea la  $O(N_1 \cdot |E|)$  folosind liste de adiacență).

Acest algoritm se folosește de următoarea proprietate: dacă la un moment dat un nod  $x$  din  $V_1$  a fost cuplat cu un nod  $y$  din  $V_2$ , nu se va decupla niciodată nodul  $x$  pentru a obține un cuplaj maxim, ci cel mult acesta va fi recuperat cu un alt nod  $z$ .

Această proprietate permite iterarea prin nodurile din  $V_1$  într-o ordine oarecare și încercarea de cuplare a fiecărui. Pentru a cupla un nod  $x$  din  $V_1$  se folosește o procedură recursivă care încearcă să cupleze  $x$  cu un nod  $y$  din  $V_2$  dacă nodul  $y$  nu este deja cuplat, sau dacă nodul cu care  $y$  este cuplat poate fi recuperat cu un nod diferit de  $y$  (lucru care se verifică apelând aceeași procedură).

Pentru a nu cicla la infinit se folosește un vector  $U$  care menține informații despre nodurile care au fost deja apelate în procedura recursivă. În implementarea prezentată mai jos se folosește o optimizare care dă rezultate foarte bune în practică: se încearcă întâi apelarea procedurii de cuplare fără a mai reseta vectorul  $U$ . Funcția de cuplare este foarte asemănătoare cu o parcurgere  $DF$ ; astfel, acest algoritm poate fi implementat și într-o manieră nerecursivă, folosind o coadă ca în parcurgerea  $BF$ .

Implementarea în limbaj a algoritmului de cuplaj, prezentată în continuare, ia în considerare următoarele declarații:

```

const MAX_N=101;
var st,dr:array[0..MAX_N]of
integer;
G:array[0..MAX_N,0..MAX_N]of
integer;
n1,n2,m,nr:integer;
u:array[0..max_n]of byte;

```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 101

int N1, N2, M, G[MAX_N][MAX_N],
st[MAX_N], dr[MAX_N], nr,
char U[MAX_N];

```

Nodurile din submulțimea  $V_1$  sunt numerotate de la 1 la  $n_1$ , iar cele din mulțimea  $V_2$ , cu numerele de la 1 la  $n_2$ .

```

1 function cupleaza(nod:integer): integer;
2   var i:integer;
3 begin
4   if u[nod]<>0 then cupleaza:=0
5   else begin
6     U[nod]:= 1;
7     for i:=1 to n2 do begin
8       if G[nod][i]=0 then
9         continue;
10      if(dr[i]=0)or(coupleaza(dr[i])<>0)
11        then begin
12          st[nod]:= i; dr[i]:= nod;
13          coupleaza:=1; break;
14        end;
15     end;
16   end;

```

```

int cupleaza(int nod)
{
  int i;
  if (U[nod]) return 0;
  U[nod] = 1;
  for (i = 1; i <= N2; i++)
  {if (!G[nod][i]) continue;
  if (!dr[i] || cupleaza(dr[i]))
  {st[nod] = i;
  dr[i] = nod;
  return 1;
  }
  }
  return 0;
}

```

```

16 procedure cuplaj;
17 var i:integer;
18 begin
19   for i:=1 to n1 do begin
20     if st[i]<>0 then continue;
21     if cupleaza(i)<>0 then inc(nr)
22     else begin
23       fillchar(u,sizeof(u),0);
24       if cupleaza(i)<>0 then inc(nr);
25     end;
26   end;
27 end;

```

### 10. Flux maxim – algoritmul lui Dinic

Fie o rețea de transport  $G=(V, E)$ , în care pentru fiecare arc există asociată o capacitate și un flux inițial nul. Considerând un vîrf  $s$  ca sursă și un vîrf  $d$  ca destinație, să se determine fluxul maxim care străbate rețeaua de la sursă la destinație.

#### Soluție:

Vom prezenta în continuare algoritmului lui Dinic de flux maxim, care are complexitate teoretică de  $O(N^2M)$  (față de  $O(N^2M^2)$  la Ford-Fulkerson). Este de menționat că în practică comportamentul acestui algoritm tinde către o complexitate  $O(N^2M)$ .

Precum Ford-Fulkerson, acest algoritm folosește o parcurgere  $BF$  pentru a construi graful rezidual asociat rețelei de transport și pentru a găsi un drum de creștere de la sursă la destinație. Diferența majoră este că nu se încearcă găsirea unui singur drum de creștere, ci se folosesc arboarele  $BF$  pentru a găsi un număr maximal (nu neapărat maxim) de drumuri de creștere. Așadar se parcurg nodurile într-o ordine oarecare, iar pentru fiecare nod  $x$  atins în parcurgerea  $BF$  se verifică dacă există o muchie în graful rezidual de la  $x$  la destinație, ca în care se verifică dacă drumul de la sursă la  $x$  din arboarele  $BF$  are capacitate reziduală  $> 0$ . Fiecare drum astfel descoperit este saturat (cu capacitatea reziduală minimă de pe drum), iar parcurgerea continuă.

Considerăm următoarele declarații:

```

const MAX_N=101;
var st,dr:array[0..MAX_N]of
integer;
G:array[0..MAX_N, 0..MAX_N]of
integer;
n1,n2,m,nr:integer;
u:array[0..max_n]of byte;

```

```

#include <stdio.h>
#include <string.h>

#define MAX_N 101
#define INF 0x3f3f
int N, M, S, D, C[MAX_N][MAX_N],
F[MAX_N][MAX_N], T[MAX_N], flux;

```

```

void cuplaj()
{
  int i;
  for (i = 1; i <= n1; i++)
  {
    if (st[i]) continue;
    if (cupleaza(i)) nr++;
    else {
      memset(U, 0, sizeof(U));
      if (cupleaza(i)) nr++;
    }
  }
}

```

Implementarea în limbaj a algoritmului lui *Dinic* este prezentată în continuare:

```

1 procedure flux_maxim;
2 var i,j,r:integer;
3 begin
4   flux:=0;
5   while bf(s,d)<>0 do
6     for i:=1 to n do begin
7       if (T[i]=-1)or(C[i][D]<=F[i][D])
8         then continue;
9       r:=C[i][D]-F[i][D];
10      j:=i;
11      while j<>s do begin
12        r:=min(r,C[T[j]][j]-F[T[j]][j]);
13        j:=T[j];
14      end;
15      if (r=0) then continue;
16      inc(F[i][D],r);
17      dec(F[D][i],r);
18      j:=i;
19      while j<>s do begin
20        j:=T[j];
21        inc(F[T[j]][j],r);
22        dec(F[j][T[j]],r);
23      end;
24      inc(flux,r);
25    end;
26  end;

```

```

void flux_maxim() {
  int i, j, r;
  for (flux = 0; BF(S, D); ) {
    for (i = 1; i <= N; i++) {
      if (T[i]==-1 || C[i][D]<=F[i][D])
        continue;
      r=C[i][D]-F[i][D];
      for (j=i; j!=S && j;
           j = T[j])
        r=min(r, C[T[j]][j]-
               F[T[j]][j]);
      if (r == 0) continue;
      F[i][D] += r;
      F[D][i] -= r;
      for (j=i; j!=S; j = T[j]){
        F[T[j]][j] += r;
        F[j][T[j]] -= r;
      }
      flux += r;
    }
  }
}

```

### 11. Flux maxim de cost minim folosind algoritmul Bellman-Ford

Fie o rețea de transport  $G=(V, E)$ , în care pentru fiecare arc există asociată o capacitate, un flux inițial nul și un cost pentru fiecare unitate de flux care trece prin acest arc. Considerând un vîrf  $s$  ca sursă și un vîrf  $d$  ca destinație, să se determine fluxul maxim de cost minim care străbate rețeaua de la sursă la destinație.

#### Soluție:

Această problemă se poate rezolva tot cu ajutorul algoritmului Ford-Fulkerson, cu condiția ca la fiecare pas să se determine un drum de creștere de cost minim. Afără unui astfel de drum este posibilă folosind algoritmul Bellman-Ford. Algoritmul Dijkstra nu se poate aplica în acest caz, deoarece acest algoritm nu funcționează în cazul existenței multihilor negative, iar în acest caz, costul arcelor introduse de algoritm va putea fi negativ (pentru arcele  $(i, j)$  din graful rezidual pe care există flux, care corespund unor arce  $(j, i)$  din graful normal).

Mentionăm că algoritmul lui *Dinic* nu poate fi modificat pentru a funcționa în acest caz. Complexitatea rezolvării este  $O(N^2M^2)$ .

Considerăm următoarele declarații:

```

const MAX_N=101;
MAX_M=1001;
INF=maxint div 2;
var n,m,s,g,flux,cost:integer;
C,F:array[0..max_n,0..max_n]of
integer;
dist,t:array[0..max_n]of integer;
muchie:array[0..max_m,0..2]of
integer;

```

Implementarea în limbaj a algoritmului de flux maxim de cost minim este prezentată în continuare:

```

1 function min(a,b:longint):longint;
2 begin
3   if a>b then min:=b
4     else min:=a;
5 end;
6
7 function bellman(sursa,dest:longint)
8           :longint;
9 var nr,ok,i,j,k,cc:longint;
10 begin
11   for i:=0 to max_n do
12     dist[i]:=inf;
13   dist[sursa]:=0;
14   ok:=1; nr:=1;
15   while (ok<>0)and(nr<N)do
16   begin
17     inc(nr);
18     ok:=0;
19     k:=0;
20     while k<M do begin
21       i:=muchie[k][0];
22       j:=muchie[k][1];
23       cc:= muchie[k][2];
24       if (C[i][j]>F[i][j])and
25         (dist[j]>dist[i]+cc)then
26       begin
27         dist[j]:=dist[i]+cc;
28         T[j]:=i;
29         ok:= 1;
30       end;
31       if (C[j][i]> F[j][i])and
32         (dist[i]>dist[j]-cc)then
33       begin
34         dist[i]:=dist[j]-cc;
35         T[i]:=j; ok:=1;
36       end;
37       inc(k);
38     end;
39     if Dist[dest]<>inf then
40       bellman:=1
41     else bellman:=0;
42   end;
43 
```

```

#include <stdio.h>
#include <string.h>
#define MAX_N 101
#define MAX_M 1001
#define INF 0x3f3f
int N, M, S, D, muchie[MAX_M][3],
C[MAX_N][MAX_N], F[MAX_N][MAX_N],
dist[MAX_N], T[MAX_N], flux,
cost;

```

```

44 procedure flux_maxim;
45 var i,r:longint;
46 begin
47   flux:=0;
48   while bellman(s,d)<>0 do
49   begin
50     r:=inf;
51     i:=d;
52     while i<>s do
53     begin
54       r:=min(r, C[T[i]][i]-
55               F[T[i]][i]);
56     i:=T[i];
57   end;
58   i:=d;
59   while i<>s do
60   begin
61     inc(F[T[i]][i],r);
62     dec(F[i][T[i]],r);
63     i:=T[i];
64   end;
65   inc(flux,r);
66   inc(cost,r*dist[D]);
67 end;
68 end;

```

## 12. Flux maxim de cost minim folosind algoritmul lui Dijkstra

Fie o rețea de transport  $G=(V, E)$ , în care pentru fiecare arc există asociată o capacitate, un flux inițial nul și un cost pozitiv pentru fiecare unitate de flux care trece prin acest arc. Considerând un vârf  $s$  ca sursă și un vârf  $d$  ca destinație, să se determine fluxul maxim de cost minim care străbate rețeaua de la sursă la destinație.

### Solutie:

După cum am menționat mai sus, algoritmul lui *Dijkstra* nu poate fi folosit în cazul în care există muchii de cost negativ în graful rezidual. Vom arăta în continuare că, plecând de la ideea că toate costurile sunt pozitive, pe parcursul iterărilor algoritmului *Ford-Fulkerson*, acestea pot fi modificate astfel încât să se poată folosi algoritmul lui *Dijkstra*, iar rezultatul furnizat să fie corect.

La prima iterătie toate costurile din graful rezidual vor fi pozitive, astfel aplicabilitatea algoritmului lui *Dijkstra* este evidentă în acest caz. Vom atribui un nou set de costuri pentru fiecare arc astfel:

$$cost'(x, y) = cost(x, y) + p(x) - p(y).$$

Scopul este ca după fiecare iterătie să atribuim valorile  $p()$  astfel încât  $cost'(x, y) \geq 0$  pentru orice arc. Trebuie remarcat că relațiile de ordine între drumurile de la sursă la restul nodurilor se păstrează, astfel încât algoritmul lui *Dijkstra* furnizează

rezultatul corect. Mai concret, fie  $d_1 = (s, a_1, a_2 \dots a_p, n)$  și  $d_2 = (s, b_1, b_2 \dots b_q, n)$  două drumuri în graf de la sursa  $s$  la nodul  $n$ ; dacă  $\text{cost}(d_1) < \text{cost}(d_2)$  atunci și  $\text{cost}'(d_1) = \text{cost}(s, a_1) + p(s) - p(a_1) + \text{cost}(a_1, a_2) + p(a_1) - p(a_2) + \dots + \text{cost}(a_p, n) + p(a_p) - p(n) = \text{cost}(d_1) + p(s) - p(n)$

$\text{cost}'(d_2) = \text{cost}(s, b_1) + p(s) - p(b_1) + \text{cost}(b_1, b_2) + p(b_1) - p(b_2) + \dots + \text{cost}(b_q, n) + p(b_q) - p(n) = \text{cost}(d_2) + p(s) - p(n)$

(lucru evident valabil și pentru  $>, =$ ).

Pentru a determina un set valid de valori  $p()$ , plecăm de la relația

$$\text{cost}'(x, y) \geq 0 \Leftrightarrow \text{cost}(x, y) + p(x) \geq p(y).$$

Fie  $d()$  distanțele minime în graful rezidual de la sursa  $s$  folosind funcția  $\text{cost}()$ . Din definiția distanțelor minime într-un graf se știe că  $\text{cost}(x, y) + d(x) \geq d(y)$ , pentru orice arc  $(x, y)$ . Tragem concluzia că  $p(x) = d(x)$ , pentru orice nod  $x$  din graf. Așadar, pentru a determina un flux maxim de cost minim folosind algoritmul lui Dijkstra se aplică următorii pași:

initializează  $p(x) = 0$  pentru fiecare nod  $x$

repetă

- construiește graful rezidual  $G_R$  asociat rețelei de transport  $G$

- găsește un drum de creștere de cost minim în graful rezidual, folosind Dijkstra și funcția de cost  $\text{cost}'()$

- incrementează  $p(x)$  cu  $d(x)$  pentru fiecare nod  $x$

- saturează acest drum

până când nu mai există drum de creștere în  $G_R$

Complexitatea algoritmului este astfel  $O(N^3 * M)$ , pentru grafuri dense, iar pentru grafuri rare se poate folosi Dijkstra cu heap-uri, pentru a obține o complexitate  $O(N * M^2 * \log N)$ .

Considerăm următoarele declarații:

```
const MAX_N=101;
MAX_M=1001;
INF=maxint div 2;
type plista^lista;
lista=record
  nod,cost:integer;urm:Plista;end;
var G:array[0..max_n]of plista;
flux,cost,n,m,s,d:integer;
U:array[0..max_n]of byte;
C,F:array[0..max_n,0..max_n]of
integer;
dist,val,T:array[0..max_n]of
integer;
```

Implementarea în limbaj a algoritmului de flux maxim de cost minim este prezentată în continuare:

```
1  function
2    min(a,b:longint):longint;
3    begin
4      if a<b then min:=a
5          else min:=b;
6    end;
7
8  function dijkstra(sursa,dest:longint)
9    :longint;
10 var i,min,nod:longint; p:plista;
11 begin
12   fillchar(U,sizeof(u),0);
13   for i:=0 to max_n do dist[i]:=inf;
14   dist[sursa]:=0;
15   while true do begin
16     min:=INF;
17     nod:=-1;
18     for i:=1 to n do
19       if (U[i]=0)and(min>dist[i])
20         then begin
21           min:=dist[i];
22           nod:=i;
23         end;
24     if min=inf then break;
25     U[nod]:=1;
26     p:=G[nod];
27     while p<>nil do begin
28       min:=dist[nod]+p^.cost+
29           val[p^.nod]-val[p^.nod];
30     if (C[p^.nod]>F[p^.nod]) and(dist[p^.nod]>min)
31       and(dist[p^.nod]>min)
32     then begin
33       dist[p^.nod]:=min;
34       T[p^.nod]:=nod;
35     end;
36     min:= dist[p^.nod]-p^.cost+
37           val[p^.nod]-val[nod];
38     if (C[p^.nod]>F[p^.nod]) and(dist[nod]>min)
39       and(dist[nod]>min) then
40     begin
41       dist[nod]:=min;
42       T[nod]:= p^.nod;
43     end;
44     p:=p^.urm;
45   end;
46   end;
47   for i:=1 to n do
48     inc(val[i],dist[i]);
49   if dist[dest]<>inf
50     then dijkstra:=1
51     else dijkstra:=0;
52 end;
53
```

```
int min(int a, int b)
{ return a < b ? a : b; }

int dijkstra(int sursa,int dest)
{
  int i, min, nod;
  lista *p;
  memset(U, 0, sizeof(U));
  memset(dist, 0x3f,sizeof(dist));
  dist[sursa] = 0;
  while (1) {
    min = INF; nod = -1;
    for (i = 1; i <= N; i++)
      if (!U[i] && min > dist[i])
        min = dist[i], nod = i;
    if (min == INF) break;
    U[nod] = 1;
    for (p=G[nod]; p; p=p->urm)
      {min=dist[nod]+p->cost+
       val[nod]-val[p->nod];
       if (C[nod][p->nod]>
       F[nod][p->nod]&&dist[p->nod]>min)
         {dist[p->nod] = min;
          T[p->nod] = nod;
        }
      }
    min=dist[p->nod]-p->cost+
    val[p->nod]-val[nod];
    if (C[p->nod][nod]>
    F[p->nod][nod]&&dist[p->nod]>min)
      {dist[nod] = min;
       T[nod] = p->nod;
      }
    }
  for (i = 1; i <= N; i++)
    val[i] += dist[i];
  return dist[dest] != INF;
}

void flux_maxim()
{int i, r;
 for (flux=0; dijkstra(S, D);
      flux+=r)
  {r = INF;
   for (i=D; i!=S; i = T[i])
     r = min(r,C[T[i]][i]-F[T[i]][i]);
   for (i=D; i!= S; i = T[i])
     {F[T[i]][i] += r;
      F[i][T[i]] -= r;
     }
   cost+=r*(val[S]-dist[S]+val[D]);
  }
}
```

```

54 procedure flux_maxim;
55 var i,x:longint;
56 begin flux:=0;
57 while dijkstra(s,d)<>0 do begin
58   x:=inf;
59   i:=d;
60   while i<>s do begin
61     x:=min(x,C[T[i]][i]-
62             F[T[i]][i]);
63     i:=t[i];
64   end;
65   i:=d;
66   while i<>s do begin
67     inc(F[T[i]][i],x);
68     dec(F[i][T[i]],x);
69     i:=T[i];
70   end;
71   inc(flux,x);
72   inc(cost,r*(val[S]-
73                 dist[S]+val[D]));
74 end;
75 end;

```

### 2.3.2 Probleme propuse

1 (\*\*). Se consideră un graf neorientat cu cel mult 20000 de muchii. Se cere să se eliminate un număr maxim de muchii astfel încât matricea închiderii tranzitive să nu se schimbe.

Fișierul *graf.in* conține, pe prima linie, numărul  $n$  ( $n < 5000$ ) de noduri, iar pe următoarele, perechi de numere  $i, j$  cu semnificația de muchie între nodurile  $i$  și  $j$ .

Fișierul *graf.out* va conține, pe prima linie, numărul  $m$  de muchii care pot fi eliminate; pe următoarele  $m$  linii, perechi de numere  $i, j$  cu semnificația că muchia dintre nodurile  $i$  și  $j$  este eliminată.

*Exemplu:*

<i>graf.in</i>	<i>graf.out</i>
6	2
1 2	2 3
2 3	5 6
3 1	
4 5	
5 6	
6 4	

2 (\*\*). Se consideră un graf neorientat cu cel mult 20000 de muchii. Se cere să se transforme, dacă este posibil, muchiile sale în arce astfel încât fiecare nod să aibă gradul extern par.

Fișierul *arce.in* conține, pe prima linie, numărul  $n$  ( $n < 5000$ ) de noduri, iar pe următoarele, perechi de numere  $i, j$  cu semnificația muchie între nodurile  $i$  și  $j$ .

Fișierul *arce.out* va conține, pe fiecare linie, perechi de numere  $i, j$  cu semnificația că muchia dintre nodurile  $i$  și  $j$  se transformă în arcul ce pleacă din  $i$  și ajunge în  $j$ .

*Exemplu*

<i>graf.in</i>	<i>graf.out</i>
4	1 2
1 2	1 4
2 3	3 2
3 4	3 4
4 1	

3 (\*\*). Se consideră un graf neorientat cu  $n$  ( $n < 1000$ ) noduri și  $m$  ( $m < 10000$ ) muchii. Pentru trei vârfuri  $a, b, c$ , să se determine dacă există un ciclu elementar care să conțină vârfurile  $a$  și  $b$  și să nu îl conțină pe  $c$ .

Fișierul *graf.in* conține, pe prima linie, numerele  $n, a, b, c$  ( $n < 5000$ ), iar pe următoarele, perechi de numere  $i, j$  cu semnificația de muchie între nodurile  $i$  și  $j$ .

Fișierul *graf.out* va conține, pe o singură linie, ciclul determinat.

*Exemplu*

<i>graf.in</i>	<i>graf.out</i>
4 2 1 4	1 2 3 1
1 2	
2 3	
3 4	
4 1	
3 1	

4 (\*\*). Într-o metropolă, nouișef al Serviciului Circulație din Poliție încearcă să micșoreze numărul de subordonăți care stau fără să aibă de lucru în birouri. În acest scop dorește să trimîtă pe teren cât mai mulți polițiști pentru a supraveghea anumite străzi ale orașului.

Pentru a-i controla mai ușor, fiecărui polițist îi va fi repartizat un traseu. Prin traseu, șeful înțelege o secvență de străzi, fiecare stradă fiind identificată prin cele două intersecții care o delimită. Stabilirea unui traseu se face în felul următor:

- din orice punct de intersecție al traseului, polițistul poate să străbată toate străzile acestuia, o singură dată, întorcându-se întotdeauna în punctul de plecare;
- în final, orice traseu repartizat unui polițist trebuie să aibă cel puțin o stradă nesupraveghetă de restul polițiștilor aflați în patrulare.

Determinați numărul maxim de polițiști care pot fi trimiși pe teren, precum și traseul repartizat fiecărui (fiecare traseu fiind scris în ordinea parcurgerii lui).

Fișierul *police.in* conține, pe prima linie, numerele  $n$  ( $n < 1500$ ) și  $m$  ( $m < 5000$ ), reprezentând numărul de intersecții, respectiv de străzi din oraș; pe următoarele  $m$  linii, perechi de două numere reprezentând numerele intersecțiilor la care este conectată fiecare stradă.

În fișierul *police.out* se va scrie, pe prima linie, numărul maxim de polițiști ( $p$ ) care pot fi trimiși pe teren; pe următoarele  $p$  linii se vor scrie traseele atribuite polițiștilor, pentru fiecare traseu precizându-se intersecțiile aflate pe acesta.

*Exemplu*

*police.in*

4 5

1 2

2 3

3 4

4 1

1 3

*police.out*

2

2 1 3 2

3 1 4 3

5 (\*\*). Un muncitor are de lăcuit toate coridoarele unui castel. El a primit o hartă în care sunt reprezentate coridoarele și cele  $n$  ( $n \leq 5000$ ) intersecții ale acestora. Unele dintre aceste intersecții sunt considerate ca fiind "punkte speciale", deoarece sunt singurele care îl permit ieșirea și intrarea în castel, prin intermediul unor uși. Muncitorului îl să-l împună să orice coridor să nu fie traversat decât o singură dată, adică atunci când este lăcuit. În aceste condiții el trebuie să lăciască toate coridoarele, folosindu-se eventual de punctele speciale, plecând din orice intersecție doarește, dar terminând operația în același punct. Realizați un program care determină un astfel de traseu ciclic pentru muncitor.

Din fișierul *castel.in* se vor citi, de pe prima linie, numerele  $n$  și  $m$  ( $m \leq 10000$ ) reprezentând numărul de intersecții și de coridoare; pe următoarele  $m$  linii, perechi de numere reprezentând intersecțiile între care există coridoare de lăcuit, iar pe ultima linie, sirul de intersecții desemnate ca puncte speciale.

În fișierul *castel.out* se vor scrie, pe o singură linie, intersecțiile, în ordinea apariției pe traseu. Numerele vor fi despărțite prin căte un spațiu sau prin caracterul \*. Perechea  $i, j$  va fi despărțită prin caracterul \* doar dacă muncitorul a ieșit din castel prin punctul special  $i$  și a revenit prin  $j$ .

*Exemplu*

*castel.in*

6 6

1 2

2 3

3 4

4 1

2 5

4 6

1 2 5 4 6

*castel.out*

1 2 3 \* 6 4 3 2 \* 4 1

6 (\*\*). Doi prieteni se află într-un oraș  $k$  și au la dispoziție  $p$  ore ( $p \leq 150$ ), timp în care să se plimbe cu mașina și apoi să ajungă fiecare la el acasă, fata în orașul  $X$ , iar băiatul în orașul  $Y$ . El doresc să se plimbe cât mai mult timp împreună, dar să ajungă în cel mult  $p$  ore, fiecare în orașul său. Mașina se deplasează cu viteză constantă. Cei doi au la dispoziție o hartă cu  $n$  orașe, legate prin străzi bidirectionale ( $3 \leq n \leq 200$ ) printre care se află, evident, orașele  $k$ ,  $X$  și  $Y$ .

Să se determine traseul comun pe care îl vor parcurge cei doi prieteni astfel încât timpul petrecut împreună să fie maxim și să ajungă amândoi la propria destinație. Cronometrarea plimbării se face începând cu momentul 0 din orașul de plecare  $k$ . În orice oraș, unul dintre ei poate continua drumul separat, cu o altă mașină.

Pe prima linie a fișierului de intrare *masina.in* se află numerele naturale  $n$ ,  $m$ ,  $k$ ,  $p$ ,  $X$ ,  $Y$  cu semnificațiile din enunț. Pe următoarele  $m$  linii sunt scrise triplete de numere naturale  $a \ b \ d$ , cu semnificația de stradă între  $a$  și  $b$  cu durată  $d$  a parcurgerii.

Rezultatele se vor scrie în fișierul *masina.out* structurate pe două linii: pe prima linie este scris un număr, reprezentând durata maximă în care cei doi se vor plimba împreună; pe linia a doua sunt scrise numerele:  $k \ x1, x2, \dots$ , reprezentând orașele pe care le vor vizita împreună, în ordine, începând cu orașul  $k$ .

*Exemplu:*

*masina.in*

4 6 1 2 0 2 4

2 1 3

4 2 1

3 1 4

4 3 3

3 2 4

4 1 2

*masina.out*

19

1 2 3 1 3 2

7 (\*\*). Se consideră o rețea de străzi reprezentată prin intersecțiile sale, numerotate de la 1 la  $n$  ( $n < 100$ ). În fiecare intersecție  $i$  există un semafor, pentru fiecare stradă  $i-j$  ( $j=1..n$ ), cu  $p$  ( $p < 50$ ) culori codificate {1, 2..,  $p$ }. Culorile semaforului au tranzițiile  $(1 \rightarrow 2), \dots, (p-1 \rightarrow p), (p \rightarrow 1)$ , iar perioada de schimbare a culorilor este 1 secundă. Un vehicul ajuns într-o intersecție  $i$  poate parcurge strada  $i-j$  tot într-o secundă. Se știe că 1 este codificarea culorii roșii care interzice, pentru orice semafor intrarea pe strada pe care o supraveghează. Realizați un program care determină un drum de durată minimă care leagă intersecțiile  $x$  și  $y$ .

Fișierul *semafor.in* conține, pe prima linie, numerele  $n$ ,  $p$ ,  $x$  și  $y$ , iar pe următoarele linii, căte trei numere  $i \ j \ c$  cu semnificația: culoarea, la momentul 0, semaforului din intersecția  $i$  care supraveghează strada  $i-j$  este  $c$ .

Fișierul *semafor.out* va conține, pe prima linie, durata traseului ales, iar pe următoarea linie intersecțiile traversate, în ordinea de pe traseu.

*Exemplu*

*semafor.in*

5 4 1 4

1 2 2

1 5 1

2 3 4

5 3 3

3 4 1

*semafor.out*

4

1 2 3 4

**8 (\*\*\*)**. Se consideră un arbore având costuri atașate pe muchii și un sir de numere naturale mai mici decât 30000. Să se adauge arborelui un număr maxim de muchii având costuri din numerele date, astfel încât graful obținut să aibă ca arbore parțial de cost minim arborele inițial.

Pe prima linie a fișierului *apm.in* se află numărul natural  $n$  reprezentând numărul nodurilor din arbore ( $0 < n \leq 256$ ). Pe următoarele  $n-1$  linii se găsesc căte trei numere naturale despărțite printr-un spațiu, sub forma  $i \ j \ c$ , având semnificația: muchia de la  $i$  la  $j$  are costul  $c$ . Pe următoarea linie se află sirul de maximum 32000 de valori naturale pozitive, mai mici ca 30000.

Pe fiecare linie a fișierului *apm.out* se vor scrie trei numere  $i \ j \ c$ , având semnificația: muchie adăugată între nodurile  $i$  și  $j$  de cost  $c$ .

*Exemplu : apm.in*

	<i>apm.out</i>
4	
1 2 2	1 4 5
3 4 4	2 3 3
3 1 2	
2 5 3	

**9 (\*\*\*)**. La o fabrică de bomboane, sunt puse pe un stand  $n$  ( $n < 101$ ) cutii de bomboane. Fiecare din cele  $n$  cutii conțin căte  $m$  ( $m < 1001$ ) bomboane dintr-un singur sortiment. Toate cele  $n$  sortimente sunt distințe. Un angajat amestecă bomboanele din cutii. Spre a nu fi observată modificarea, el are grija ca în fiecare cutie să rămână căte  $m$  bomboane. Din păcate, șeful său îi cere să-i aducă căte o bomboană din fiecare cutie, deci din fiecare sortiment. Fiind urmărit, muncitorul nu poate extrage decât o bomboană din fiecare cutie.

Muncitorul acordă fiecărei extrageri un grad de risc. Astfel, la extragerea unei bomboane din sortimentul cu numărul  $i$  din cutia care conținea inițial sortimentul cu numărul  $j$ , gradul de risc va fi valoarea absolută a diferenței dintre  $i$  și  $j$ . Gradul de risc al tuturor celor  $n$  extrageri va fi egal cu suma riscurilor fiecăreia.

Să se determine ce sortiment de bomboană va trebui să extragă muncitorul din fiecare cutie, pentru ca gradul de risc total să fie minim.

Pe prima linie a fișierului *bombon.in* sunt scrise numerele  $n$  și  $m$ . Pe următoarele  $n$  linii se găsesc  $m$  numere care reprezintă sortimentele bomboanelor ce se regăsesc în fiecare cutie după amestecare, în ordine, începând cu cutia având numărul de ordine 1 până la cutia având numărul de ordine  $n$ .

Fișierul de ieșire *bombon.out* conține, pe prima linie, un număr reprezentând gradul total minim de risc. Pe a doua linie sunt scrise  $n$  numere reprezentând numărul de ordine al cutiei din care va fi scoasă bomboana din fiecare sortiment, în ordine, începând cu sortimentul 1 la  $n$ .

*Exemplu : bombon.in*

	<i>bombon.out</i>
4 5	
1 1 2 4 4	2
2 4 2 3 4	1 2 4 3
1 2 4 1 3	
2 1 3 3 3	

**10 (\*\*\*\*)**. Fie o rețea de transport  $G=(V, E)$ , în care, pentru fiecare arc, există asociată o capacitate inferioară, o capacitate superioară și un flux inițial nul. Considerând un vârf  $s$  ca sursă și un vârf  $d$  ca destinație, să se determine dacă poate exista în această rețea, și în caz afirmativ să se determine fluxul maxim care străbate rețeaua de la sursă la destinație și respectă constrângările.

**11 (\*\*\*\*)**. Fie  $G=(V,E)$ , graf orientat cu costuri pozitive întregi asociate arcelor. Să se determine o submulțime de muchii de cost minim astfel încât să nu mai existe nici un drum de la un nod  $s$  dat la un alt nod  $d$  dat.

**12 (\*\*\*)**. Fie  $G=(V,E)$ , graf orientat. Să se determine numărul minim de muchii care trebuie eliminate pentru ca graful să devină neconex.

**13 (\*\*)**. Fie o rețea de transport  $G=(V, E)$ , în care pentru fiecare nod există asociată o capacitate și un flux inițial nul. Considerând un vârf  $s$  ca sursă și un vârf  $d$  ca destinație, să se determine fluxul maxim care străbate rețeaua de la sursă la destinație.

**14 (\*\*\*\*)**. O acoperire cu drumuri a unui graf aciclic orientat  $G = (V, E)$  este o mulțime de drumuri disjuncte astfel încât fiecare vârf din  $V$  este inclus în exact un drum din această mulțime. Drumurile pot începe și se pot termina oriunde și pot avea orice lungime, inclusiv 0. Să se determine o acoperire cu drumuri minimă.

**15 (\*\*\*\*)**. Într-un magazin există  $T \leq 100$  tipuri de unelte, fiecare având ca preț un număr pozitiv. Pentru o listă dată de  $P \leq 100$  proiecte (pentru fiecare proiect se știe profitul care se obține efectuându-l, cât și uneltele necesare) să se determine profitul maxim (costul proiectelor efectuate minus costul uneltelelor cumpărate) care se poate obține. O unelă se cumpără o singură dată și poate fi folosită pentru mai multe proiecte.

**16 (\*\*)**. *Gigel* este pasionat de informatică, și mai ales de cifrele 0 și 1, așa de mult, încât a atribuit fiecărui dintre cei  $2^N$  prietenii ai lui câte o etichetă, sub forma unui sir de biți de lungime  $N \leq 20$ . Toate etichetele sunt distințe între ele.

*Gigel* s-a gândit într-o zi că vrea să construiască o etichetă pentru el însuși, de lungime cât mai mică, care să conțină o singură dată, ca o subsecvență, fiecare dintre cele  $2^N$  etichete ale prietenilor lui. Scrieți un program care determină eticheta lui *Gigel*, de lungime minimă.

Pe prima linie a fișierului de intrare *biti.in* se va găsi numărul  $N$ .

Pe prima linie a fișierului de ieșire *biti.out* se va găsi lungimea sirului. Pe a doua linie se va afișa un sir de biți 0 sau 1 care va reprezenta eticheta găsită.

*Exemplu*

<i>biti.in</i>	<i>biti.out</i>
3	10 0001011100

17 (\*\*\*) . Gigel s-a mutat într-un oraș nou! Pentru a se familiariza cu noile împrejurimi a cumpărat harta orașului și a observat că este alcătuită din  $M$  străzi de diferite lungimi, cu sens unic și  $N \leq 60$  intersecții de străzi. Gigel a luat harta și a început să alcătuiască un traseu care pornește dintr-o intersecție anume, trece prin fiecare stradă cel puțin o dată, și revine în intersecția de unde a pornit. Deși dorința lui de explorare este mare, condiția lui fizică nu este tocmai bună, astfel că vrea să găsească un traseu în care suma lungimilor străzilor parcuse este minimă.

Pe prima linie din fișierul *traseu.in* se găsesc numerele  $N$  și  $M$  separate prin căte un spațiu. Pe următoarele  $M$  linii se vor găsi triplete de numere  $i \ j \ k$  cu semnificația că există o stradă de la intersecția cu număr  $i$  la intersecția cu număr  $j$  de lungime  $k$ . Lungimile străzilor sunt numere naturale din intervalul  $[1, 10.000]$ .

Pe prima linie din fișierul *traseu.out* se va afișa un singur număr natural, reprezentând lungimea minimă a traseului lui Gigel.

*Exemplu*

*traseu.in*

```
6 8
1 2 3
2 3 1
3 1 2
1 4 4
4 6 2
6 1 5
4 5 1
5 1 6
```

*traseu.out*

```
28
```

18 (\*\*\*) . Se dă un graf orientat cu  $n \leq 100.000$  noduri și  $m \leq 200.000$  arce, fiecare arc având o pondere  $k \leq 1000$ . Se dau două noduri  $x$  și  $y$ . Se cere determinarea valorii  $k$  minime, astfel încât să existe un drum de la nodul  $x$  la nodul  $y$  cu proprietatea că fiecare arc de pe drum are ponderea  $k_i$  mai mică sau egală decât  $k$ . Între nodurile  $x$  și  $y$  va exista întotdeauna un drum.

Fișierul *pscny.in* va conține, pe prima linie, patru numere întregi ce reprezintă valorile lui  $n$ ,  $m$ ,  $x$  și  $y$ . Pe următoarele  $m$  linii se vor afla căte trei numere întregi  $x_b$ ,  $y_b$  și  $k_b$ ,  $x_i$  fiind nodul din care pleacă arcul,  $y_i$  nodul în care ajunge și  $k_i$  reprezentând ponderea arcului.

Fișierul *pscny.out* va conține un singur număr întreg ce reprezintă valoarea minimă a lui  $k$ .

*Exemplu : pscny.in*

```
4 5 1 4
1 2 3
1 3 1
2 4 2
3 2 2
3 4 3
```

*pscny.out*

```
2
```

19 (\*\*\*) . Fiind dat un graf planar cu  $N \leq 10.000$  noduri și  $M \leq 60.000$  muchii, aflați numărul maxim de noduri pe care îl poate avea un subgraf complet al său (un graf neorientat se numește complet dacă există muchie între oricare două noduri ale sale). De asemenea se cere și numărul de subgrafuri complete, cu număr maxim de noduri, care se găsesc în graful planar dat.

Prima linie a fișierului de intrare *count.in* conține două numere naturale  $N$  și  $M$  (numărul de noduri, respectiv numărul de muchii ale grafului planar). Următoarele  $M$  linii conțin căte două numere  $A$  și  $B$ , cu semnificația: există o muchie bidirectională între nodurile  $A$  și  $B$  (nodurile grafului fiind numerotate de la 1 la  $N$ ).

Fișierul de ieșire *count.out* va conține, pe prima linie, două numere  $X$  și  $Y$  reprezentând numărul maxim de noduri pe care îl poate avea un subgraf complet și, respectiv, numărul de subgrafuri complete cu  $X$  noduri din graful planar dat.

*Exemplu : count.in*

```
5 8
1 2
1 3
1 5
2 4
2 5
3 4
3 5
4 5
```

*count.out*

```
3 4
```

20 (\*\*\*) . Un bun prieten de-al lui Zăhărel, Bronzărel este la spital. Fiind buni prieteni, Zăhărel se duce să-l viziteze pe acesta. Ultima dată când a fost în vizită, Bronzărel desenase pe peretei spitalului mai multe grafuri neorientate conexe cu costuri (da, și Bronzărel este pasionat de informatică!) și mai mult de atât, calculase, pentru fiecare dintre acestea, distanțele minime de la un nod sursă ales la toate celelalte. Zăhărel, curios ca întotdeauna, a decis că vrea să vadă dacă Bronzărel delirează sau nu și și-a notat pe o foaie de hârtie ceea ce scriise acesta pe perete. Când a ajuns acasă, el s-a decis să vadă pentru care din grafuri distanțele minime calculate de Bronzărel erau corecte.

Scripteți un program care-l ajută pe Zăhărel să determine, pentru fiecare graf, dacă distanțele minime au fost calculate corect.

Pe prima linie din fișierul de intrare *distante.in* se va găsi un număr  $T \leq 10$ , reprezentând căte grafuri sunt pe foaie. Următoarele linii vor descrie succesiv căte un graf. Prima linie din descrierea unui graf va conține numerele  $N \leq 50.000$ ,  $M \leq 100.000$ , și  $S$ , reprezentând numărul de noduri, numărul de muchii și nodul

sursă de la care se calculează distanțele minime. A doua linie din descriere conține  $N$  elemente  $D_1, D_2, \dots, D_N$  reprezentând distanțele minime calculate de *Bronzare*. Următoarele  $M$  linii vor conține căte trei numere naturale  $a, b, c$  reprezentând faptul că există o muchie de cost  $c$  de la  $a$  la  $b$ .

Fișierul de ieșire *distanțe.out* va conține  $T$  linii, fiecare cu căte un cuvânt: "DA", dacă distanțele minime au fost calculate corect pentru graful respectiv sau "NU", altfel.

*Exemplu: distante.in*

	<i>distanțe.out</i>
2	
5 6 1	DA
0 1 7 3 6	NU
1 2 1	
1 3 7	
1 4 3	
3 4 4	
2 5 5	
4 5 6	
4 4 2	
1 0 2 3	
1 2 1	
2 3 1	
2 4 1	
3 4 1	

## CAPITOLUL 3

### Metode de programare

#### 3.1. Metoda backtracking

##### 3.1.1 Probleme rezolvate

###### 1. Elemente de combinatorică – Generarea permutărilor unui sir de valori

Se citește de la tastatură un sir de  $n$  ( $n \leq 10$ ) numere naturale distincte, de cel mult trei cifre. Să se realizeze un program care afișează, în fișierul *perm.out*, toate permutările sirului dat, fiecare pe căte o linie. Numerele vor fi despărțite în cadrul liniei prin căte un spațiu.

*Exemplu*

Pentru  $n=3$  și sirul de valori: 3,6,8

Fișierul *perm.out* va conține:

3 6 8  
3 8 6  
6 3 8  
6 8 3  
8 6 3  
8 3 6

*Soluție:*

Notăm cu  $A$  vectorul care memorează sirul dat. Algoritmul de generare a permutărilor, folosind metoda backtracking, utilizează o stivă (*sol*) în care se vor construi soluțiile, bazându-se pe următoarea observație: pe orice nivel în stiva *sol* poate fi așezat orice element din sir, neutilizat pe nivelurile anterioare.

Deducem astfel că soluțiile corecte (permutările) aparțin produsului cartezian  $A^N$ .

*Codificarea datelor:*

- Cele  $n$  elemente ale sirului vor fi memorate în vectorul  $A$ ;
- Elementele vectorului  $Sol(N)$  vor reține indicii elementelor din  $A$ . Astfel,  $Sol[k]=i$  are semnificația: elementul  $A[i]$  este așezat în permutare pe poziția  $k$ .

*Condițiile interne ale soluției:*

- $Sol[i] \neq Sol[j]$ , pentru orice pereche  $(i, j)$  unde  $1 \leq i, j \leq n$  și  $i \neq j$ .

*Condițiile de continuare:*

- $Sol[k] \neq Sol[i]$ , unde  $1 \leq i \leq k-1$  și  $k \leq n$ .

Strategia de generare a permutărilor unei mulțimi A poate fi transcrisă în pseudocod în felul următor:

```

1 backtracking( A[], sol[], k)
2   daca k > |A| atunci afiseaza
3   altfel
4     pentru i = 1, |A| execută
5       sol[k] ← i
6       daca asezare_corecta(k) atunci
7         backtracking(k+1)
8       altfel
9         ...
10    ...

```

În cadrul implementării algoritmului prezentat, funcția *ok* verifică dacă toate condițiile de continuare sunt îndeplinite. Funcția returnează valoarea *True/1* dacă sunt îndeplinite condițiile, respectiv *False/0*, în caz contrar.

```

1 var n,i:integer;
2 a,sol:array[0..10]of integer;
3
4 function ok(k:byte):boolean;
5 var i:integer;
6 begin
7   ok:=true;
8   for i:=1 to k-1 do
9     if sol[k]=sol[i] then
10      ok:=false;
11 end;
12
13 procedure afis();
14 var i:integer;
15 begin
16   for i:=1 to n do
17     write(a[sol[i]],' ');
18   writeln();
19 end;
20
21 procedure back(k:byte);
22 var i:integer;
23 begin
24   if k>n then afis()
25   else
26     for i:=1 to n do begin
27       sol[k]:=i;
28       if ok(k) then back(k+1);
29     end;
30 end;
31
32 begin
33   assign(output,'perm.out');
34   rewrite(output); readln(n);
35   for i:=1 to n do read(a[i]);
36   back(1);
37 end.

```

Varianta următoare de implementare a algoritmului renunță la funcția *ok* și utilizează un vector *Use* cu elemente aparținând mulțimii {*True/False*}, respectiv {0/1}, în varianta C++. În cadrul acesteia, elementul *Use[i]=True* (*Use[i]=1*), dacă elementul *A[i]* este așezat în permutare, (indicele *i* se află printre elementele vectorului *Sol*) și *False/0*, în caz contrar.

```

1 procedure back(k:byte);
2 var i:integer;
3 begin
4   if k>n then afis()
5   else
6     for i:=1 to n do
7       if not use[i] then begin
8         sol[k]:=i; u
9         use[i]:=true;
10        back(k+1);
11        use[i]:=false;
12      end;
13 end;

```

```

void back(int k) {
  int i;
  if (k>n) afis();
  else {
    for (i=1; i<=n; i++)
      if (!use[i]) {
        sol[k]=i;
        use[i]=1;
        back(k+1);
        use[i]=0;
      }
  }
}

```

## 2. Elemente de combinatorică – Aranjamente de *n* obiecte luate câte *m*

Se consideră o mulțime de *n* ( $n \leq 10$ ) numere naturale. Să se realizeze un subprogram care generează toate grupurile de *m* elemente distincte ale mulțimii date, grupuri care diferă fie prin valorile elementelor, fie prin ordinea lor (aranjamente de *n* luate câte *m*).

*Exemplu:*

Pentru  $n=3, m=2$  și sirul de valori:  
3, 6, 8

Se vor genera:  
3 6  
3 8  
6 3  
6 8  
8 6  
8 3

*Soluție:*

Notăm cu *A* vectorul care memorează elementele mulțimii date. Algoritmul de generare a aranjamentelor este asemănător cu cel de la permutări. Diferența constă în lungimea soluțiilor. Deducem astfel că soluțiile corecte (aranjamentele) aparțin produsului cartezian  $A^M$ .

*Codificarea datelor:*

- cele *n* elemente ale sirului vor fi memorate în vectorul *A*;
- elementele vectorului *Sol(M)* vor reține indicii elementelor din *A*. Astfel, *Sol[k]=i* are semnificația: elementul *A[i]* este așezat în aranjament pe poziția *k*;
- elementul *Use[i]=True/1*, dacă elementul *A[i]* este așezat în aranjament (indicele *i* se află printre elementele vectorului *Sol*) și *False/0*, în caz contrar.

*Condițiile interne ale soluției:*

- *Sol[i] ≠ Sol[j]*, pentru orice pereche  $(i, j)$  unde  $1 \leq i, j \leq m$  și  $i \neq j$ .

Implementarea subprogramului de generare și în considerare următoarele declarații:

```
var n,m,i:integer;
a,sol:array[0..10] of integer;
use:array[0..10] of Boolean;
```

Prezentăm în continuare subprogramul care generează aranjamentele de  $m$  obiecte luate câte  $m$ :

```
1 procedure back(k:byte);
2 var i:integer;
3 begin
4 if k>m then afis
5 else {
6 for i:=1 to n do
7 if not use[i] then begin
8 sol[k]:=i;
9 use[i]:=true;
10 back(k+1);
11 use[i]:=false;
12 end;
13 end;
14 }
```

```
void back(int k) {
    int i;
    if (k>m) afis();
    else {
        for (i=1; i<=n; i++) {
            if (!use[i]) {
                sol[k]=i;
                use[i]=1;
                back(k+1);
                use[i]=0;
            }
        }
    }
}
```

### 3. Elemente de combinatorică – Combinări de $n$ obiecte luate câte $m$

Se consideră o mulțime formată din  $n$  ( $n \leq 10$ ) numere naturale. Să se realizeze un program care generează toate submulțimile de  $m$  elemente ale mulțimii date (combinări de  $n$  luate câte  $m$ ). Soluțiile vor fi scrise în fișierul *comb.out*.

*Exemplu:*

Pentru  $n=3$ ,  $m=2$  și sirul de valori:

3, 6, 8

Fișierul *comb.out* va conține:

3 6  
3 8  
6 8

Soluție

Notăm cu  $A$  vectorul care memorează sirul dat. Algoritmul de generare a permutărilor, folosind metoda backtracking, utilizează o stivă (*sol*) în care se vor construi soluțiile, bazându-se pe următoarea observație: pentru a evita permutarea elementelor în cadrul soluțiilor, pe orice nivel, în stiva *sol*, se va așeza un element al cărui indice este strict mai mare decât al elementului așezat pe nivelul anterior.

*Codificarea datelor:*

- Cele  $n$  elemente ale sirului vor fi memorate în vectorul  $A[1..n]$ ;
- Elementele vectorului  $Sol(M)$  vor reține indicii elementelor din  $A$ . Astfel,  $Sol[k]=i$  are semnificația: elementul  $A[i]$  este așezat în combinare pe poziția  $k$ ;

*Condițiile interne ale soluției:*

- $Sol[k-1] < Sol[k]$ , pentru orice indice  $k$ ,  $1 < k \leq m$ .

Implementarea în limbaj al algoritmului de generare a combinărilor de  $n$  obiecte luate câte  $m$  este prezentată în continuare:

```
1 var m,n,i:integer;
2 a,sol:array[0..10] of integer;
3
4 procedure afis;
5 var i:integer;
6 begin
7 for i:=1 to m do
8   write(a[sol[i]], ' ');
9 writeln;
10 end;
11
12 function ok(k:byte):boolean;
13 begin
14   ok:=true;
15   if (k>1)and(sol[k]<=sol[k-1])
16     then ok:=false;
17 end;
18
19 procedure back(k:byte);
20 var i:integer;
21 begin
22   if k>m then afis
23   else {
24     for i:=1 to n do begin
25       sol[k]:=i;
26       if ok(k) then back(k+1);
27     end;
28   end;
29
30 begin
31   assign(output, 'comb.out');
32   rewrite(output); readln(n,m);
33   for i:=1 to n do read(a[i]);
34   back(1);
35 end.
```

```
#include <stdio.h>

int n, m, i, a[11], sol[11];

void afis() {
    int i;
    for (i=1; i<=m; i++)
        printf("%d ", a[sol[i]]);
    printf("\n");
}

int ok(int k) {
    if (k>1&&sol[k]<=sol[k-1])
        return 0;
    return 1;
}

void back(int k) {
    int i;
    if (k>m) afis();
    else {
        for (i=1; i<=n; i++) {
            sol[k]=i;
            if (ok(k)) back(k+1);
        }
    }
}

int main() {
    freopen("comb.out", "w", stdout);
    scanf("%d %d", &n, &m);
    for(i=1;i<=n;i++) scanf("%d",&a[i]);
    back(1);
    return 0;
}
```

Dacă se renunță la funcția *ok* de validare a condițiilor de continuare, implementarea subprogramului devine:

```
1 procedure back(k:byte);
2 var i:integer;
3 begin
4 if k>m then afis
5 else {
6   for i:=sol[k-1]+1 to n do
7     begin
8       sol[k]:=i;
9       back(k+1);
10      end;
11    end;
12  end;
```

```
void back(int k)
{
    int i;
    if (k>m) afis();
    else {
        for(i=sol[k-1]+1;i<=n;i++) {
            sol[k]=i;
            back(k+1);
        }
    }
}
```

#### 4. Elemente de combinatorică – Generarea submulțimilor unei mulțimi de n obiecte

Se consideră un sir de  $n$  ( $n \leq 10$ ) numere naturale distincte. Să se realizeze un program care generează toate submulțimile nevide ale unei mulțimi date. Soluțiile vor fi scrise în fișierul *subm.out*.

*Exemplu:*

Pentru  $n=2$  și sirul de valori: 3, 6

Fișierul *subm.out* va conține:  
3  
3:6  
6

*Soluție:*

Notăm cu *A* vectorul care memorează sirul dat. Algoritmul de generare a submulțimilor este asemănător cu cel de la combinări. Diferența constă în lungimea soluțiilor care variază între 1 și  $n$ . Numărul soluțiilor, adică al submulțimilor nevide ale unei mulțimi de  $n$  obiecte, este  $2^n - 1$ .

Afișarea unei submulțimi se va face după fiecare așezare în vectorul *Sol* a unui nou element ce respectă condițiile de continuare.

*Codificarea datelor:*

- Cele  $n$  elemente ale sirului vor fi memorate în vectorul *A*;
- Elementele vectorului *Sol(N)* vor reține indicii elementelor din *A*. Astfel, *Sol[k]=i* are semnificația: elementul *A[i]* este așezat în submulțime pe poziția *k*.

*Condițiile interne ale soluției:*

- $Sol[k-1] < Sol[k]$ , pentru orice indice  $k$ ,  $1 < k \leq n$ .

Implementarea în limbaj a algoritmului de generare a submulțimilor unei mulțimi de  $n$  obiecte este prezentată în continuare:

```

1  var n,i:integer;
2  a,sol:array[0..10] of integer;
3
4  procedure afis(k:byte);
5  var i:integer;
6  begin
7    for i:=1 to k do
8      write(a[sol[i]],' ');
9    writeln;
10   end;
11
12  procedure back(k:byte);
13  var i:integer;
14  begin
15    if k<=n then
16      for i:=sol[k-1]+1 to n do
17        begin
18          sol[k]:=i;
19          afis(k);
20          back(k+1);
21        end;
22    end;
23
24  begin
25    assign(output,'subm.out');
26    rewrite(output);
27    readln(n);
28    for i:=1 to n do read(a[i]);
29    back(1);
30  end.

```

```

int main() {
  freopen("subm.out", "w", stdout);
  scanf ("%d", &n);
  for (i=1;i<=n;i++)scanf ("%d",&a[i]);
  back(1);
  return 0;
}

```

#### 5. Elemente de combinatorică – Generarea partițiilor unei mulțimi de $n$ obiecte

Se consideră o mulțime *A* formată din  $n$  valori întregi. Să se realizeze un program care generează toate partițiile mulțimii date. Prin partitie se înțelege o descompunere a mulțimii inițiale într-o reuniune de mulțimi disjuncte. Soluțiile vor fi scrise în fișierul *part.out*.

*Exemplu:*

Pentru  $n=3$  și sirul de valori: 3, 6, 8

Fișierul *part.out* va conține:  
{3 6 8}  
{3}{6 8}  
{3 8}{6}  
{3 6}{8}  
{3}{6}{8}

*Soluție:*

Notăm cu *A* vectorul care memorează sirul dat. Algoritmul de generare va contoriza numărul de submulțimi existente în cadrul partitiei. Fiecare element al vectorului *A* poate fi așezat în partitie în oricare din cele *nr* submulțimi generate până la acel moment sau într-o nouă submulțime, având numărul de ordine *nr+1*.

*Codificarea datelor:*

- Cele  $n$  elemente ale sirului vor fi memorate în vectorul *A*;
- Variabila *nr* reține numărul de submulțimi existente în partitie;
- Elementele vectorului *Sol(N)* vor reține numerele de ordine ale submulțimilor partitiei. Astfel, *Sol[k]=i* are semnificația: elementul *A[k]* este așezat în submulțimea cu numărul de ordine *i*;

*Condițiile interne ale soluției:*

- $Sol[k] \leq i + 1$ , unde  $1 \leq i \leq nr$ , pentru orice indice  $k$ ,  $1 \leq k \leq n$ .

În aceste condiții, generarea nu necesită validarea condițiilor de continuare. Implementarea în limbaj, a algoritmului de generare a partițiilor unei mulțimi de  $n$  obiecte este prezentată mai jos:

```

1  var n,i:integer;
2  a,sol:array[0..10] of integer;
3
4  procedure afis(nr:byte);
5  var i,j:integer;
6  begin
7    include <stdio.h>
8    int n, i, a[11], sol[11];
9
10   procedure afis(int k)
11   {
12     int i;
13     for (i=1; i<=k; i++)
14       printf("%d ", a[sol[i]]);
15     printf("\n");
16   }
17
18   void back(int k)
19   {
20     int i;
21     if (k<=n)
22       for(i=sol[k-1]+1;i<=n;i++) {
23         sol[k]=i;
24         afis(k);
25         back(k+1);
26       }
27   }
28
29  begin
30    assign(output,'part.out');
31    rewrite(output);
32    readln(n);
33    for i:=1 to n do read(a[i]);
34    back(1);
35  end.

```

```

6 begin
7   for i:=1 to nr do begin
8     write(' ');
9     for j:=1 to n do
10       if sol[j]=i then write(a[j], ' ');
11     write(' ');
12   end;
13   writeln;
14 end;

15 procedure back(k,nr:byte);
16 var i:integer;
17 begin
18   if k>n then afis(nr)
19   else begin
20     for i:=1 to nr do begin
21       sol[k]:=i;
22       back(k+1,nr);
23     end;
24     sol[k]:=nr+1;
25     back(k+1,nr+1);
26   end;
27 end;
28
29 begin
30   assign(output,'part.out');
31   rewrite(output); readln(n);
32   for i:=1 to n do read(a[i]);
33   back(1,0);
34 end.

```

## 6. Numere

Să se genereze toate numerele de  $n$  cifre ( $n \leq 6$ ) care nu conțin trei cifre consecutive pe poziții alăturate și ale căror cifre se află în ordine strict descrescătoare. Fiecare număr rezultat va fi scris pe căte o linie în fișierul text *out.txt*.

### Soluție:

#### Codificarea datelor:

- Elementele vectorului  $Sol(N)$  vor reține cifrele numerelor generate, de la 0 la 9. Astfel,  $Sol[k]=i$  are semnificația: pe poziția  $k$ , în reprezentarea zecimală a numărului, se află cifra  $i$ ;

#### Condițiile interne ale soluției:

- $0 \leq Sol[k] \leq 9$ , pentru orice indice  $k$ ,  $1 \leq k \leq n$ ;
- Pe nivelul  $k$  cifrele care pot constitui valori corecte aparțin mulțimii  $\{0..Sol[k-1]-1\}$  sau în situația în care, pe ultimele două niveluri construite ( $k-1$  și  $k-2$ ), se află deja cifre consecutive, mulțimea este  $\{0..Sol[k-1]-2\}$ . Se ține cont și de ordinea descrescătoare de apariție a cifrelor.

```

for (i=1;i<=nr;i++) {
  printf(" ");
  for (j=1;j<=n;j++)
    if (sol[j]==i)printf("%d",a[j]);
  printf("\n");
}

void back(int k,int nr) {
  int i;
  if (k>n) afis(nr);
  else {
    for (i=1;i<=nr;i++) {
      sol[k]:=i; back(k+1,nr);
    }
    sol[k]:=nr+1; back(k+1,nr+1);
  }
}

int main() {
  freopen("part.out", "w", stdout);
  scanf("%d", &n);
  for (i=1;i<=n;i++)scanf("%d",&a[i]);
  back(1, 0);
  return 0;
}

```

O astfel de construcție a soluțiilor nu mai necesită validarea cifrei așezate pe nivelul curent. Subprogramul este implementat în manieră recursivă.

```

1. war n:integer; f:text;
2. sol:array[0..10] of integer;
3.
4. procedure back(k:integer);
5. var i:integer;
6. begin
7.   if k<=n then
8.     if (0<2)&&(sol[k-2]==sol[k-1]+1)
9.     then
10.       for i:=sol[k-1]-2 downto 0 do
11.         begin
12.           sol[k]:=i;
13.           back(k+1);
14.         end
15.       else
16.         for i:=sol[k-1]-1 downto 0 do
17.           begin
18.             sol[k]:=i;
19.             back(k+1);
20.           end
21.       else begin
22.         for i:=1 to n do write(f,sol[i]);
23.         writeln(f);
24.       end;
25.     end;
26.   begin
27.     assign(f,'out.txt');
28.     rewrite(f); readln(n);
29.     sol[0]:=10; back(1);
30.     close(f);
31.   end.

```

```

# include <stdio.h>
int sol[100],n;

void back(int ik)
{
  int i;
  if (ik<=n)
    if (ik>2&&sol[ik-2]==sol[ik-1]+1)
      for (i=sol[ik-1]-2;i>=0;i--)
        {sol[ik]=i;
        back(ik+1);}
    else
      for (i=sol[ik-1]-1;i>=0;i--)
        {sol[ik]=i;
        back(ik+1);}
  else
    for (i=1;i<=n;i++)
      printf("%d",sol[i]);
    printf("\n");
  return;
}

void main()
{
  freopen("out.txt","w",stdout);
  printf("%d",n);
  scanf("%d",&n);
  sol[0]=10;
  back(1);
  fclose(stdout);
}

```

## 7. Grupuri

Se consideră un sir de  $n$  numere întregi. Să se genereze submulțimile de  $m$  numere din sir în care se regăsesc cel puțin trei numere care pot reprezenta laturile unui triunghi. Fiecare combinație obținută va fi scrisă pe căte o linie în fișierul text *out.txt*, numerele în cadrul liniei fiind separate prin căte un spațiu.

### Soluție:

#### Codificarea datelor:

- Cele  $n$  elemente ale sirului vor fi memorate în vectorul  $A$ ;
- Elementele vectorului  $Sol(M)$  vor reține indicii elementelor din  $A$ . Astfel,  $Sol[k]=i$  are semnificația: elementul  $A[i]$  este așezat în submulțime pe poziția  $k$ .

#### Condițiile interne ale soluției:

- $Sol[k-1] < Sol[k]$  pentru orice indice  $k$ ,  $1 < k \leq n$ .

Pentru a afișa o soluție este necesară verificarea existenței unui triplet de elemente din sir care pot constitui laturile unui triunghi (suma oricărui două numere din cele trei este mai mare decât a treia).

Implementarea în limbajul algoritmului de generare este realizată în manieră recursivă:

```

1  var i,m,k,n:integer; f:text;
2  sol,a:array[0..100]of integer;
3
4  procedure afis;
5  begin
6    for i:=1 to m do
7      write(f,a[sol[i]],' ');
8      writeln(f);
9    end;
10
11 function T:boolean;
12 var i,j,k,x,y,z:integer;
13 begin
14   T:=false;
15   for i:=1 to m-2 do
16     for j:=i+1 to m-1 do
17       for k:=j+1 to m do begin
18         x:=sol[i];
19         y:=sol[j]; z:=sol[k];
20         if (a[x]<a[y]+a[z]) and
21             (a[y]<a[x]+a[z]) and
22             (a[z]<a[x]+a[y]) then
23           T:=true;
24       end;
25     end;
26
27 procedure back(k:integer);
28 var i:integer;
29 begin
30   if k=m+1 then begin
31     if T then afis end
32   else
33     for i:=sol[k-1]+1 to n do
34     begin
35       sol[k]:=i;
36       back(k+1);
37     end;
38   end;
39
40 begin
41   assign(f,'out.txt');
42   rewrite(f);
43   readln(n,m);
44   for i:=1 to n do read(a[i]);
45   sol[0]:=0;
46   back(1);
47   close(f);
48 end.

```

```

#include <stdio.h>
#include <iostream>
int a[100],sol[100],n,m;

void afis()
{
    int i;
    for(i=1;i<=m;i++)
        printf("%d ",a[sol[i]]);
    printf("\n");
    return;
}

int T()
{
    int x,y,z,s,i,j,k;
    s=0;
    for(i=1;i<=m-2;i++)
        for(j=i+1;j<=m-1;j++)
            for(k=j+1;k<=m;k++)
            {
                x=sol[i];y=sol[j];z=sol[k];
                if ((a[x]<a[y]+a[z])&&
                    (a[y]<a[x]+a[z])&&
                    (a[z]<a[x]+a[y]))
                    s+=1;
            }
    return s;
}

void back(int k)
{
    int i;
    if (k==m+1)
        { if (T()) afis(); }
    else
        for(i=sol[k-1]+1;i<=n;i++)
        {
            sol[k]:=i;
            back(k+1);
        }
    return;
}

void main(void)
{
    int i;
    freopen("out.txt","w",stdout);
    scanf("%d%d",&n,&m);
    for(i=1; i<=n; i++)
        scanf("%d",&a[i]);
    sol[0]=0;
    back(1);
    fclose(stdout);
}

```

### 8. Note muzicale

Să se genereze toate sirurile de  $n$  ( $n \leq 6$ ) note muzicale din mulțimea  $\{do, re, mi, fa, sol, la, si\}$ . Fiecare sir va fi afișat pe căte o linie în fișierul text *out.txt*, notele din cadrul liniei fiind separate prin căte un spațiu. Acestea se pot repeta în cadrul unui sir.

#### Soluție:

Algoritmul realizează generarea  $n$ -tupletelor produsului cartezian  $X^N$ , unde  $X=\{1..7\}$ .

#### Codificarea datelor:

- Notele muzicale sunt reținute prin intermediul tabloului  $A$  cu elemente de tip sir de caractere;
- Elementele vectorului  $Sol(N)$  vor reține indicii elementelor din  $A$ . Astfel,  $Sol[k]=i$  are semnificația: nota muzicală  $A[i]$  este așezată în sirul soluție pe poziția  $k$ .

Nu există condiții interne pe care trebuie să le îndeplinească elementele din soluție. Subprogramul de generare a soluțiilor este implementat în varianta recursivă.

```

1  const
2  a:array[1..7]of string=
3  {'do','re','mi','fa','sol',
4  'la','si'};
5  var i,n,k:integer; f:text;
6  sol:array[1..6] of integer;
7
8  procedure afis;
9  begin
10   for i:=1 to n do
11     write(f,a[sol[i]],' ');
12     writeln(f);
13   end;
14
15 procedure back(k:integer);
16 var i:integer;
17 begin
18   if k=n+1 then afis
19   else
20     for i:=1 to 7 do begin
21       sol[k]:=i;
22       back(k+1);
23     end;
24   end;
25
26 begin
27   write('n='); readln(n);
28   assign(f,'out.txt'); rewrite(f);
29   back(0); close(f);
30 end.

```

```

#include <stdio.h>
#include <iostream>
int sol[100],n;
char* a[7]=
{"do","re","mi","fa","sol",
"la","si"};

void afis()
{
    int i,j;
    for(i=0;i<=n-1;i++)
        printf("%s ",a[sol[i]]);
    printf("\n");
    return;
}

void back(int k)
{
    int i;
    if (k==n) afis();
    else
        for (i=0;i<=6;i++)
        {
            sol[k]:=i;
            back(k+1);
        }
    return;
}

void main(void)
{
    freopen("out.txt","w",stdout);
    scanf("%d",&n);
    assign(f,'out.txt'); rewrite(f);
    back(0);
    fclose(stdout);
}

```

### 9. Intervale

Se consideră n intervale de numere reale identificate prin limita inferioară și prin cea superioară  $(a_i, b_i)$ . Să se genereze toate secvențele de intervale cu proprietatea că oricare două intervale din secvență au intersecția vidă. Fiecare soluție se va scrie în fișierul *out.txt* în formatul  $(a_1, b_1) (a_2, b_2) \dots$

*Exemplu:*

Pentru  $n=3$  și intervalele

(2, 5.5),  
 (3, 6),  
 (6.7, 10)

Fisierul *out.txt* va conține:  
(2,5,5)  
(3,6)  
(6,7,10)  
(2,5,5)(6,7,10)  
(3,6)(6,7,10)

Solutie:

Fiecare interval este codificat în cadrul unei date de tip înregistrare care conține capătul stâng și capătul drept ale acestuia. În vederea generării tuturor submulțimilor de intervale cu reuniune vidă s-a optat pentru ordonarea crescătoare a intervalelor în funcție de capătul stâng al acestora.

### *Codificarea datelor:*

- Vectorul  $A$  reține sirul de înregistrări ce reprezintă cele  $n$  intervale;
  - În vectorul  $Sol$  se vor construi submulțimile soluție. Elementele vectorului  $Sol(N)$  vor reține indicii elementelor din  $A$ . Astfel,  $Sol[k]=i$  are semnificația intervalul  $A[i]$  este așezat în submulțimea soluție pe poziția  $k$ .

#### **Condițiile interne ale soluției:**

- $Sol[k-1] < Sol[k]$ , pentru orice indice  $k$ ,  $1 < k \leq n$

*Condiții de continuare:*

- Elementul  $Sol[k]$ , ( $k \leq n$ ) este considerat valid dacă limita stângă a intervalului  $A[Sol[k]]$  este mai mare decât capătul drept al intervalului  $A[Sol[k-1]]$ .

Implementarea algoritmului este prezentată în continuare

```

1 type interval=record
2   cs,cd:real;end;
3 var n,i:integer;
4   a:array[1..100]of interval;
5   sol:array[0..100]of byte;
6
7 procedure sort;
8 var i,j:integer; aux:interval;
9 begin
10   for i:=1 to n-1 do
11     for j:=i+1 to n do
12       if a[i].cs>a[j].cs then begin
13         aux:=a[i]; a[i]:=a[j];
14         a[j]:=aux; end;
15   end;

```

```
#include <stdio.h>
struct inter{
    float cs,cd; }a[100];
int sol[100],n;

void scrie(int k)
{
    int i;
    for(i=1;i<=k;i++) {
        printf("%5.1f",a[sol[i]].cs);
        printf("%5.1f",a[sol[i]].cd);
    }
    printf("\n");
    return;
}
```

```

16. procedure scrie(k:integer);
17. var i:integer;
18. begin
19.   for i:=1 to k do begin
20.     write('(' ',a[sol[i]].cs:5:1);
21.     write(' ',a[sol[i]].cd:5:1,')');
22.   end;
23.   writeln;
24. end;
25.
26. procedure back(k:integer);
27. var i:integer;
28. begin
29.   if k<=n then
30.     for i:=sol[k-1]+1 to n do
31.       if (a[i].cs>a[sol[k-1]].cd)
32.         then begin
33.           sol[k]:=i;
34.           scrie(k);
35.           back(k+1);
36.         end
37.   end;
38.
39. begin
40.   readln(n);
41.   for i:=1 to n do
42.     readln(a[i].cs,a[i].cd);
43.   sort();
44.   sol[0]:=0;
45.   back(1);
46. end.

```

---

```

void sort(void)
{int i,j, inter aux;
for(i=1;i<n;i++)
  for(j=i+1;j<=n;j++)
    if (a[i].cs>a[j].cs)
      {aux=a[i];
       a[i]=a[j];
       a[j]=aux;
      }
}

void back(int k)
{int i;
if (k<=n)
  for (i=sol[k-1]+1;i<=n;i++)
    if (a[i].cs>a[sol[k-1]].cd)
      {sol[k]:=i;
       scrie(k);
       back(k+1);}
  return;
}

void main(void)
{int i;
scanf("%d",&n);
for(i=1;i<=n;i++)
scanf("%f%f",&a[i].cs,&a[i].cd);
sort();
sol[0]=0;
back(1);
}

```

10. Sesiune

Un student are de susținut într-o sesiune  $n$  examene, numerotate cu numere de la 1 la  $n$ , în funcție de ordinea în care trebuie să fie susținute. Sesiunea durează un număr de  $m$  zile consecutive ( $n \leq [m/2] \leq 10$ ), iar studentul poate să-și programeze fiecare examen în ce zi dorește, respectând doar ordinea acestora. În plus, el dorește să nu aibă de susținut două examene în zile consecutive. Să se genereze toate programările pe zile ale examenelor în condițiile dorite de student. Pe fiecare linie a fișierului text *out.txt* vor fi scrise câte  $n$  numere reprezentând zilele în care se vor susține în ordine, cele  $n$  examene. Zilele sesiunii vor fi codificate prin numere de la 1 la  $m$ .

### *Exemplum*

Considerăm  $m=5$  și  $n=$

| Fisierul *out.txt* va contine

13  
14  
15  
24  
25  
35

### Soluție:

Algoritmul trebuie să genereze combinațiile de  $m$ -elemente ale mulțimii  $\{1..m\}$ , ale căror elemente respectă condiția impusă prin enunț (nu există două elemente consecutive).

### Codificarea datelor:

- Elementele vectorului  $Sol(N)$  vor reține valori din mulțimea  $\{1..m\}$ . Astfel,  $Sol[k]=i$  are semnificația: examenul cu numărul de ordine  $k$  este susținut în ziua numărul  $i$ ;

### Condițiile interne ale soluției:

- $Sol[k-1] < Sol[k]$ , pentru orice indice  $k$ ,  $1 < k \leq m$ ;
- $Sol[k] - Sol[k-1] \geq 2$ , pentru orice indice  $k$ ,  $1 < k \leq m$ ;

Implementarea în limbaj al algoritmului este prezentată în continuare:

```

1 var n,m:integer; f:text;
2 sol:array[0..100]of shortint; #include <stdio.h>
3
4 procedure scrie;
5 var i:integer;
6 begin
7   for i:=1 to n do
8     write(f,sol[i],',');
9   writeln(f);
10 end;
11
12 procedure back(k:integer);
13 var i:integer;
14 begin
15   if k<=n then
16     for i:=sol[k-1]+2 to m do begin
17       sol[k]:=i;
18       back(k+1);
19     end
20   else scrie;
21 end;
22
23 begin
24   readln(n,m);
25   assign(f,'out.txt'); rewrite(f);
26   sol[0]:=-1;
27   back(1);
28   close(f);
29 end.
  
```

### 11. Subșir

Se consideră un sir de  $n$  ( $n \leq 10$ ) numere naturale mai mici decât 30000. Să se genereze toate subșirurile crescătoare de  $m$  numere care se pot forma începând cu primul element din sir. Fiecare subșir rezultat va fi scris în fișierul *out.txt* pe căte o linie.

### Exemplu:

Pentru  $n=6$ ,  $m=3$  și numerele:  
3 1 5 6 4 8

Fișierul *out.txt* va conține:

3 4 5  
3 4 6  
3 4 8  
3 5 6  
3 5 8  
3 6 8

### Soluție:

Pentru optimizarea generării soluțiilor s-a optat pentru sortarea inițială a elementelor din sir și regăsirea noii poziții pe care se află primul element.

### Codificarea datelor:

- Elementul  $Sol[1]$  reține noua poziție pe care se află primul element în urma sortării;
- Celelalte  $m-1$  elemente ale vectorului  $Sol$  rețin valori din mulțimea  $\{Sol[j], n\}$ . Astfel,  $Sol[k]=i$  are semnificația: pe poziția  $k$  din sirul soluție se află elementul  $A[i]$  (după sortarea lui  $A$ ).

### Condițiile interne ale soluției:

- $Sol[k-1] < Sol[k]$ , pentru orice indice  $k$ ,  $1 < k \leq m$ .

Implementarea în limbaj al algoritmului este prezentată în continuare:

```

1 var n,i,m,x:integer; f:text;
2 sol,a:array[0..100]of byte; #include <stdio.h>
3
4 procedure sort;
5 var i,j,aux:integer;
6 begin
7   (... sortare vector a)
8 end;
9
10 function cauta(x,p,u:integer):byte;
11 var m:integer;
12 begin
13   m:=(p+u) div 2;
14   if x=a[m] then cauta:=m
15   else if p<=u then
16     if x< a[m] then
17       cauta:=cauta(x,p,m-1)
18     else cauta:=cauta(x,m+1,u)
19   else cauta:=0;
20 end;
21
22 procedure scrie;
23 var i:integer;
24 begin
25   for i:=1 to m do
26     write(f,a[sol[i]],',');
27   writeln(f);
28 end;
  
```

```

29 procedure back(k:integer);
30 var i:integer;
31 begin
32 if k<m+1 then
33   for i:=sol[k-1]+1 to n do
34   begin
35     sol[k]:=i;
36     back(k+1);
37   end
38 else scrie;
39 end;
40
41 begin
42   readln(n,m);
43   assign(f,'out.txt'); rewrite(f);
44   for i:=1 to n do read(a[i]);
45   x:=a[1];
46   sort();
47   sol[1]:=cauta(x,1,n);
48   back(2);
49   close(f);
50 end.

```

```

void back(int k)
{
int i;
if (k<m+1)
  for (i=sol[k-1]+1; i<=n; i++)
    { sol[k]=i;
      back(k+1); }
else scrie();
return;
}

void main(void)
{
int i,x;
scanf ("%d%d", &n, &m);
freopen("out.txt", "w", stdout);
for (i=1; i<=n; i++)
  scanf ("%d", &a[i]);
x=a[1];
sort();
sol[1]= cauta(x,1,n);
back(2);
fclose(stdout);
}

```

## 12. Mouse

Pe o tablă dreptunghiulară cu  $n$  linii și  $m$  coloane ( $n, m \leq 10$ ), există zone libere marcate cu 0 și zone cu obstacole marcate cu 1. Știind că pe tablă se află un șoricel la poziția  $x, y$  și o bucată de brânză la poziția  $z, t$ , să se găsească toate traseele pe care le poate străbate șoricelul până la brânză. Acesta se poate mișca pe tablă în oricare zonă alăturată (pe linie sau coloană) poziției curente, unde nu se află un obstacol și nu poate repeta o poziție a tablei în cadrul unui traseu.

Din fișierul *mouse.in* se va citi, în ordine, de pe prima linie numerele  $n, m, x, y, z, t$ , iar de pe următoarele  $n$  linii codificarea tablei.

Traseele determinate se vor afișa pe ecran, pe câte o linie, în formatul din exemplu.

*Exemplu:*

*mouse.in*  
4 6 3 1 1 6  
1 0 0 0 0 0  
0 0 1 1 0 1  
0 0 0 0 0 1  
1 1 1 1 1 1

Se va afișa:  
(3,1)(2,1)(2,2)(1,2)(1,3)(1,4)(1,5)(1,6)  
(3,1)(3,2)(2,2)(1,2)(1,3)(1,4)(1,5)(1,6)  
(3,1)(3,2)(3,3)(3,4)(3,5)(2,5)(1,5)(1,6)  
(3,1)(2,1)(2,2)(3,2)(3,3)(3,4)(3,5)(2,5)(1,5)(1,6)

*Solutie:*

Algoritmul de generare simulează toate mișările posibile ale șoricelului din fiecare poziție validă de pe tablă. Considerând o astfel de poziție  $(x,y)$ , există 4 elemente succesoare acesteia.

Pentru a evita validarea suplimentară a unei poziții (dacă se află sau nu în interiorul tablei) s-a optat pentru "bordarea" matricei, cu obstacole pe liniile 0 și  $n+1$ , respectiv coloanele 0,  $m+1$ .

### Codificarea datelor:

- Variabila globală  $k$  reține numărul de poziții de pe traseul curent.
- Tabloul  $Dr$  simulează stiva în care se rețin traseele. Elementele  $Dr[1, j]$  și  $Dr[2, j]$  rețin indicii poziției cu numărul  $j$  de pe drum ( $1 \leq j \leq k$ ).
- Matricea  $A$  reține configurația tablei. În plus, elementul  $A[i, j]=2$ , dacă poziția este deja plasată pe traseul curent.
- Tablourile  $dx$  și  $dy$  sunt folosite pentru identificarea indicilor vecinilor unei poziții de pe tablă.

### Condiții de continuare:

- Poziția curentă trebuie să fie în interiorul tablei, fără obstacol, și neplasată deja pe traseu:  $A[x, y]=0$ .

Considerăm următoarele declarații:

```

const
dx:array[1..4] of integer=(-1,0,1,0);
dy:array[1..4] of integer=(0,1,0,-1);
var
a:array[0..10,0..10] of integer;
j,n,m,x,y,k,c,d,q,r,z,t:integer;
dr:array[1..2,1..100] of integer;

```

```

#include <stdio.h>
const int dx[] = {-1, 0, 1, 0};
const int dy[] = {0, 1, 0, -1};
int k,n,m,x,y,z,t, a[11][11],
dr[2][101];

```

Subprogramul *back()*, care generează traseele posibile, este prezentat în continuare:

```

1 procedure citire;
2 var i,j:integer;
3 begin ...citire date... end;
4
5 procedure bordare;
6 begin
7   for c:=0 to n+1 do begin
8     a[c,0]:=1; a[c,m+1]:=1;
9   end;
10  for c:=0 to m+1 do begin
11    a[0,c]:=1; a[n+1,c]:=1;
12  end;
13 end;
14
15 procedure afis;
16 begin ...afisare drum... end;
17
18 procedure back(x,y:integer);
19 var i:integer;
20 begin
21   if (x=z) and (y=t) then afis
22   else
23     if a[x,y]=0 then begin
24       a[x,y]:=2; inc(k);
25       dr[1,k]:=x; dr[2,k]:=y;
26       for i:=1 to 4 do
27         back(x+dx[i], y+dy[i]);
28       a[x,y]:=0; dec(k);
29     end;
30 end;

```

```

void citire() {
  // citire date
}

void bordare() {
  int c;
  for (c=0; c<=n+1; c++)
    a[c][0]=a[c][m+1]=1;
  for (c=0; c<=m+1; c++)
    a[0][c]=a[n+1][c]=1;
}

void afis() {
  // afisare drum
}

void back(int x, int y) {
  int i;
  if (x==z&&y==t) {
    afis();
    return;
  }
  if (a[x][y]==0) {
    a[x][y]=2; k++;
    dr[0][k]=x; dr[1][k]=y;
    for (i=0;i<4;i++)
      back(x+dx[i], y+dy[i]);
    a[x][y]=0;
    k--;
  }
}

```

```

31 begin :citire;
32 bordare;
33 assign(output, "out.txt");
34 rewrite(output);
35 back(x,y);
36 close(output);
37 end.

```

```

int main()
{
    :citire();
    bordare();
    freopen("out.txt", "w", stdout);
    back(x,y);
    return 0;
}

```

### 3.1.2 Probleme propuse

- Să se genereze toate sirurile de lungime  $m$  ( $1 \leq m \leq 10$ ) formate din divizori ai unui număr natural  $x$  ( $1 \leq x \leq 999$ ), aflați în ordine crescătoare. Fiecare sir rezultat va fi scris pe căte o linie în fișierul text *sir.out*, iar numerele de pe linie vor fi despărțite prin căte un spațiu. În cazul în care nu există soluție se va scrie mesajul ‘‘Imposibil’’.
- Să se genereze toate cuvintele de lungime  $n$  ( $n \leq 10$ ) ale alfabetului Morse (formate doar din caracterele ‘‘.’’ și ‘‘-’’) care nu încep și nu se termină cu caracterul ‘‘-’’. Fiecare cuvânt va fi afișat pe căte o linie în fișierul text *morse.out*.
- Se consideră sirul cifrelor de la 0 la 9. Să se genereze toate numerele de  $n$  cifre ( $n \leq 9$ ) care nu conțin trei cifre pare sau trei cifre impare alăturate. Fiecare număr rezultat va fi scris pe căte o linie în fișierul text *cifre.out*.
- Să se genereze toate tablourile pătratice binare de ordin  $n$  ( $n \leq 10$ ), care au proprietatea că atât fiecare linie, cât și fiecare coloană conțin exact un element egal cu 1. Fiecare tablou va fi afișat pe  $n$  linii în fișierul text *matrice.out*, numerele în cadrul liniei fiind separate prin căte un spațiu. În fișier, între tablourile rezultante va exista o linie goală.

*Exemplu:*

Pentru  $n=2$

```

matrice.out
0 1
1 0

```

- Se citesc de la tastatură numerele naturale  $n$  ( $n \leq 10$ ) și  $m$  ( $m \leq 50$ ). Considerăm primele  $m$  numere prime de cel puțin două cifre. Să se scrie în fișierul, *prime.out*, toate sirurile de  $n$  numere prime distincte alese din cele  $m$ , fiecare sir scris pe căte o linie.

*Exemplu:*

Pentru  $n=2$  și  $m=3$

```

prime.out
11 13
13 11
11 17
17 11
13 17
17 13

```

- Profesorul de sport are să așeze elevii în ordinea crescătoare a înălțimii lor. Cunoscând numele și înălțimea fiecărui elev dintre cei  $n$  ( $n \leq 10$ ), să se scrie în fișierul *sport.out* variantele pe care le are profesorul. Fiecare sir corect va fi scris pe căte o linie, numele copiilor fiind despărțite prin căte un spațiu.

*Exemplu:*

Pentru  $n=4$  și elevii

```

maria 1:65
ion 1:60
mihai 1:78
ana 1:65

```

```

sport.out
ion maria ana mihai
ion ana maria mihai

```

- Se citesc de la tastatură numerele naturale  $n$  ( $n \leq 10^9$ ) și  $m$  ( $m \leq 9$ ). Considerăm sirul primelor  $m$  cifre distincte ale lui  $n$  (cele mai mici). Să se scrie, pe o singură linie, în fișierul *nr.out* toate numerele, la cărora scriere în baza 10 folosește numai cifre din sirul primelor  $m$ . Numerele vor fi scrise în fișier în ordine crescătoare, despărțite prin căte un spațiu. Ele nu vor conține mai mult de  $m$  cifre.

*Exemplu:*

Pentru  $n=48311378$  și  $m=2$

Sirul primelor  $m$  cifre distincte ale lui  $n$  este format din {1, 3} iar fișierul *nr.out* conține numerele:  
1 3 11 13 31 33

- Se citesc de la tastatură numerele naturale  $n$  ( $n \leq 10^9$ ) și  $S$  ( $S \leq 45$ ). Să se afișeze toate submulțimile de sumă  $S$ , formate din cifrele lui  $n$ . Fișierul text *subm.out* va conține, pe căte o linie, cifrele dintr-o submulțime generată.

*Exemplu:*

Pentru  $n=211357$  și  $S=10$

```

subm.out
2 3 5
3 7
2 1 7

```

- Să se genereze toate numerele naturale care sunt reprezentate în baza 2 prin  $m$  ( $m \leq 9$ ) cifre, dintre care  $p$  cifre de 1. Valorile ce constituie soluții vor fi afișate pe ecran pe o singură linie.

*Exemplu:*

Pentru  $m=4$  și  $p=2$

Se va afișa:  
9 10 12

- Considerăm  $n$  ( $n \leq 10$ ) tipuri de monede. Din fiecare tip avem la dispoziție un număr nelimitat de monede. Generați toate modalitățile de plată a unei sume  $S$  ( $S \leq 30000$ ), în condițiile date. În fișierul *monede.out* vor fi afișate variantele, în formatul din exemplu.

*Exemplu:*

Pentru  $S=5$ ,  $n=4$  și valorile monedelor:

```

2, 8, 1, 4

```

Se va afișa:  
5\*1  
3\*1 + 1\*2  
1\*1 + 1\*4  
1\*1 + 2\*2

11. Considerăm un sir de  $n$  ( $n \leq 10$ ) cifre. Plasati în fața oricărei valori, amul din operatorii  $+$ ,  $-$ ,  $*$ , astfel încât expresia rezultată să aibă valoarea  $S$ . Cifrele nu își vor schimba pozițiile în cadrul sirului. Afisați pe ecran, pe câte o linie, expresiile obținute.

*Exemplu:*

Pentru  $S=5$ ,  $n=4$  și valorile :

Se va afișa:  
+2 +8 -1 -4

12. Se consideră un sir de  $n$  ( $n \leq 10$ ) litere mici distincte. Să se genereze toate cuvintele care verifică următoarele condiții:

- un cuvânt conține exact  $m$  litere din sirul celor  $n$ ;
- literele din care este format, apar în ordine lexicografică (crescătoare).

Datele de intrare se citesc de la tastatură, iar soluțiile vor fi scrise în fișierul de ieșire *sir.out*, pe o singură linie, despărțite prin câte un spațiu. Ele vor forma un sir ordonat lexicografic crescător.

*Exemplu:*

Pentru  $n=4$ ,  $m=2$  și sirul de litere:

Fișierul *sir.out* va conține:  
ab ac ad bc bd cd

13. Se consideră o listă formată din  $n$  ( $n \leq 10$ ) cuvinte. Să se construiască cu ele cel mai lung sir în care fiecare cuvânt folosit în sir are ultima literă mai mică în sens lexicografic decât prima literă a cuvântului următor.

Datele de intrare se citesc din fișierul de tastatură, iar sirul maximal va fi scris în fișierul de ieșire *sir.out*.

*Exemplu:*

Pentru  $n=4$  și cuvintele:  
ana, capat, repede, merge

Fișierul *sir.out* va conține:  
anarepedemerge

14. Se consideră un sir de  $n$  ( $n \leq 10$ ) cuvinte și un număr  $m$  ( $m \leq 20$ ) natural. Se cere determinarea celui mai lung cuvânt format prin alipirea cuvintelor date în care nici o literă nu apare de mai mult de  $m$  ori. Nu contează ordinea în care sunt concatenate cuvintele.

Datele de intrare se citesc de tastatură, iar sirul maximal va fi scris în fișierul de ieșire *sir.out*.

*Exemplu:*

Pentru  $n=5$ ,  $m=3$  și cuvintele:  
cand, poate, frunza, pica, toamna

Fișierul *sir.out* va conține:  
poatefrunzacand

15. Considerăm un vector  $A$  ce conține  $n$  ( $n \leq 10$ ) numere întregi. Vom spune că două elemente ale sale formează o "pereche în dezordine" dacă sunt îndeplinite simultan condițiile:

- $i < j$ , unde  $1 \leq i < n$  și  $1 < j \leq n$ ;
- $a[i] > a[j]$ .

Să se creeze un program care generează toate sirurile care conțin același număr de perechi în dezordine ca și vectorul inițial. Datele de intrare vor fi citite de la tastatură. Fișierul text *perechi.out* va conține pe câte o linie câte un sir generat.

*Exemplu:*

Pentru  $n=4$  și vectorul  $A=(1,3,2,4)$

Fișierul *perechi.out* va conține:

1243  
2134

16. Considerăm un vector  $A$  ce conține  $n$  ( $n \leq 10$ ) numere întregi. Să se genereze toate permutările acestuia care respectă condiția că cel mai mic și cel mai mare număr să rămână pe pozițiile inițiale. Datele se introduc de la tastatură, iar soluțiile obținute vor fi afișate pe ecran, câte una pe fiecare linie.

*Exemplu:*

Pentru  $n=4$  și vectorul  $A=(3,2,3,2,1)$

Se va afișa:  
3 2 3 2 1  
2 2 3 3 1

17. Se consideră un număr natural  $n$  ( $n \leq 100$ ). Să se determine toate modurile de scriere a lui ca sumă de numere prime. Soluțiile obținute vor fi afișate pe ecran, câte una pe fiecare linie.

*Exemplu:*

Pentru  $n=14$

Printre soluțiile afișate se vor afla:  
2 2 2 2 2 2 2  
3 3 3 3 2  
3 3 3 5  
3 11  
7 7

18. Se consideră o tablă de săh de dimensiune  $n \times m$  ( $n, m \leq 10$ ). Să se determine toate modurile de așezare a  $n$  dame pe tablă, astfel încât acestea să nu se atace reciproc (linie, coloană sau diagonală).

19. Considerăm un sir de  $n$  ( $n \leq 10$ ) scaune. Pe acestea au fost așezăți copii identificați prin numele lor. Determinați toate modalitățile de reașezare a copiilor pe scaune, astfel încât nici unul dintre ei să nu aibă aceeași vecini ca în aranjamentul inițial. Soluțiile obținute vor fi afișate pe ecran, câte una pe fiecare linie, iar cuvintele vor fi despărțite prin câte un spațiu.

*Exemplu:*

Pentru  $n=4$  și aranjamentul inițial:  
Ana, Maria, Ion, Vlad

Se va afișa:  
Maria Vlad Ana Ion  
Ion Ana Vlad Maria

20. Pentru un număr natural par  $n$  ( $n \leq 10$ ), se cere realizarea unui program care generează toate sirurile de paranteze rotunde care se închid corect. Soluțiile se vor afișa pe ecran.

*Exemplu:*

Pentru  $n=6$  se va afișa: (000, (0)0, 0(0), (00), ((0))

21. Considerăm codificarea unei fotografii alb-negru într-o matrice binară cu  $n$  linii și  $m$  coloane ( $n, m \leq 7$ ). Obiectele conțin puncte negre codificate cu valoarea 1 în matrice, iar culoarea albă cu 0. Două elemente egale cu 1 fac parte din același obiect dacă sunt vecine pe linie, coloană sau diagonală. Colorați obiectele din fotografie cu numere de la 2 la  $nr+1$  ( $nr$  reprezintă numărul de obiecte din fotografie). Din fișierul *foto.in* se citesc, de pe prima linie, numerele  $n$  și  $m$ , iar de pe următoarele  $n$  linii, codificarea fotografiei. În fișierul *foto.out* se va scrie matricea obținută în urma procesului de colorare.

*Exemplu:*

*foto.in*

4 5  
0 1 0 0 1  
1 0 0 0 1  
0 0 1 0 0  
0 1 1 1 0

<i>foto.out</i>	0 2 0 0 3 2 0 0 0 3 0 0 4 0 0 0 4 4 4 0
-----------------	--

22. Considerăm o hartă a unei regiuni muntoase, codificate sub formă unei matrice cu  $n$  linii și  $m$  coloane ( $n, m \leq 7$ ). Un alpinist se află în punctul de coordonate  $x, y$  și vrea să părăsească zona muntoasă. El se poate deplasa în orice zonă vecină pe linie sau coloană, în care diferența de altitudine nu este mai mare decât o valoare  $h$  cunoscută, fără să treacă de două ori prin aceeași zonă. Realizați un program care determină traseele pe care le poate urma alpinistul.

Din fișierul *harta.in* se vor prelua de pe prima linie, în ordine, numerele  $n$ ,  $m$ ,  $h$ ,  $x$ ,  $y$ , iar de pe următoarele linii, codificarea hărții. În fișierul *harta.out* se vor scrie traseele determinante, în formatul din exemplu.

*Exemplu:*

*harta.in*  
3 5 10 2 2  
13 14 27 98 15  
32 21 82 33 32  
43 23 24 32 65

<i>harta.out</i>	(2,2)(1,2) (2,2)(3,2)(3,3)(3,4)(2,4)(2,5)
------------------	--

23. Un râu urmează să fie traversat de  $n$  persoane având greutățile  $g_1, g_2, \dots, g_n$ . Barca pe care o au la dispoziție, poate suporta o greutate de maximum  $G_{max}$  kilograme. Vâslașul se consideră a avea greutate neglijabilă. Să se determine o modalitate de a îmbarca persoanele, astfel încât să se efectueze cât mai puține transporturi. Datele se citesc de la tastatură și soluția se va afișa pe ecran, în formatul din exemplu:

*Exemplu:*

Pentru  $n=4$ ,  $G_{max}=100$  și greutățile:  
40, 98, 2, 59

{40,59}{98,2}

## 3.2. Metoda Divide et Impera

### 3.2.1 Probleme rezolvate

#### 1. Maxim

Se consideră un sir de  $n$  numere întregi memorate cu ajutorul unui vector  $A$ . Să se realizeze o funcție care returnează valoarea maximă din sir, folosind metoda *Divide et Impera*. Subprogramul va primi prin parametrii întregi  $l, r$ , limitele subtabloului unde se caută elementul maxim.

#### Soluție:

Metoda de programare *“Divide et Impera”* poate fi aplicată în cazul problemelor a căror soluție se poate obține prin descompunerea în subprobleme similare de dimensiuni mai mici. Procesul de descompunere este repetat, până la obținerea unor subprobleme cu rezolvare imediata. Trebuie însă menționat că operația de descompunere trebuie să genereze *subprobleme independente*, adică subprobleme ce prelucrează mulțimi disjuncte de date de intrare. O dată determinate soluțiile acestora, ele vor fi combinate pentru obținerea soluției problemei inițiale. Datorită structurii sale, strategia metodei *Divide-et-Impera* este implementată recursiv.

Strategia de rezolvare a problemei urmărește trei etape:

1. *Descompunerea problemei curente* într-un număr de două probleme similare, de dimensiune mai mică. Astfel, dacă subtabloul curent în care se caută maximul, conține mai mult de un element, atunci problema se descompune în două subprobleme similare, una care cauță maximul în subtabloul delimitat de indicii  $(l, m)$ , iar cealaltă, în subtabloul delimitat de  $(m+1, r)$ . Variabila  $m$  reține indicele de mijloc al subtabloului curent.

Dacă dimensiunea problemei curente este suficient de mică, în cazul de față, conține un singur element, rezolvarea ei nu mai necesită o nouă descompunere, iar soluția o reprezintă elementul de indice  $l$ . În această ultimă situație, s-a ajuns la *cazul de bază* al procesului recursiv.

2. *Rezolvarea subproblemelor obținute* prin descompunerea problemei curente. Această rezolvare este posibilă prin continuarea procesului recursiv de descompunere, până la obținerea unor subprobleme cu rezolvare imediata.

3. *Combinarea soluțiilor subproblemelor*, pentru obținerea soluției problemei curente și, în final, a celei inițiale. În cazul de față, se compară soluțiile celor două subprobleme obținute în procesul de descompunere, iar valoarea maximă reprezintă soluția pe intervalul delimitat de indicii  $(l, r)$ .

```

1 Divide_et_Impera(A[], left, right)
2   daca left = right atunci solutia ← A[left]
3   altfel
4     middle ← ⌊(left + right)/2⌋
5     sol1 ← Divide_et_Impera(left, middle) //rezolv subprob 1
6     sol2 ← Divide_et_Impera(middle + 1, right) //rezolv subprob 2
7     daca sol1>sol2 atunci
8       solutia ← sol1
9     altfel
10    solutia ← sol2
11
12

```

Există situații când faza de combinare a soluțiilor lipsește. Este cazul în care soluția problemei inițiale:

- reprezintă soluția unei subprobleme obținute în urma descompunerii;
- este construită simultan cu etapele de descompunere.

Prezentăm în continuare implementarea în limbaj, a subprogramului cerut:

```

1 function max(l, r:integer):
2   integer;
3 var m,x,y:integer;
4 begin
5   if l = r then max:=a[l]
6   else begin
7     m:=(l+r)div 2;
8     x:=max(l,m);
9     y:=max(m+1, r);
10    if x > y then max := x
11    else max := y;
12  end; end;
13
14 int max(int l, int r)
15 {
16   int m,x,y;
17   if (l==r)
18     return a[l];
19   m=(l+r)/2;
20   x=max(l,m);
21   y=max(m+1, r);
22   if x > y ? x : y;
23 }

```

## 2. Cel mai mare divizor comun

Se consideră un sir de  $n$  numere întregi memorate cu ajutorul unui vector A. Să se realizeze o funcție care returnează cel mai mare divizor comun al celor  $n$  valori, folosind metoda *Divide et Impera*. Subprogramul va primi prin parametrii întregi  $l, r$ , limitele subtabloului pentru ale cărui elemente se determină c.m.m.d.c.

Soluție:

1. **Descompunerea problemei curente:** Dacă subtabloul curent delimitat de indicii  $(l, r)$  conține mai mult de două elemente, atunci problema se descompune în două subprobleme similare, una care determină c.m.m.d.c al elementelor din subtabloul delimitat de indicii  $(l, m)$ , iar cealaltă în subtabloul delimitat de  $(m+1, r)$ . Variabila  $m$  reține indicele de mijloc al subtabloului curent.

Dacă  $r-l \leq 1$ , problema nu mai necesită o nouă descompunere, iar soluția o reprezintă cmmdc( $A[l], A[r]$ ).

2. **Rezolvarea subproblemelor obținute** prin descompunerea problemei curente. Această rezolvare este posibilă prin continuarea procesului recursiv de descompunere, până la obținerea unor subprobleme cu rezolvare imediată.

3. **Combinarea soluțiilor subproblemelor:** În cazul de față, se aplică asociativitatea operației c.m.m.d.c.:

$$c.m.m.d.c(a_1, \dots, a_r) = c.m.m.d.c(c.m.m.d.c(a_1, \dots, a_m), c.m.m.d.c(a_{m+1}, \dots, a_r))$$

Funcția Euclid() returnează c.m.m.d.c-ul valorilor parametrilor întregi  $x$  și  $y$ , folosind algoritmul lui Euclid, prin împărțiri succesive.

```

1 function euclid(x, y:integer):
2   integer;
3 var x:integer;
4 begin
5   while y <> 0 do begin
6     r := x mod y;
7     x := y;
8     y := r;
9   end;
10  euclid :=x;
11 end;
12
13 function cmmdc(l, r:integer):
14   integer;
15 var m:integer;
16 begin
17   if r - l <= 1 then
18     cmmdc:=euclid(a[l], a[r]);
19   else begin
20     m:=(l+r)div 2;
21     cmmdc:=euclid(cmmdc(l,m),
22                     cmmdc(m+1,r));
23   end;
24 end;

```

```

int euclid(int x,int y)
{
  int r;
  while (y)
  { r=x%y;
    x=y;
    y=r;
  }
  return x;
}

int cmmdc(int l,int r)
{
  int m;
  if (r-l<=1)
    return euclid(a[l], a[r]);
  m=(l+r)/2;
  return euclid(cmmdc(l,m),
                cmmdc(m+1,r));
}

```

## 3. Rădăcina pătrată întreagă

Considerăm un număr natural  $n$ . Rădăcina pătrată întreagă a lui  $n$  este, prin definiție, un număr natural  $p$ , pentru care  $p \leq \sqrt{n} < p + 1$ . Fără să folosiți funcția predefinită radical, realizați un subprogram care determină numărul  $p$ , folosind metoda *Divide et Impera*.

Soluție:

1. **Descompunerea problemei curente:** Problema inițială determină valoarea lui  $p$  în interiorul intervalului  $[0, n+1]$ . Generalizând, presupunem că limitele intervalului de căutare sunt  $l$  și  $r$ . Această problemă se descompune în două subprobleme similare, una care cauță rădăcina în intervalul  $[l, m]$ , iar cealaltă, în intervalul  $[m, r]$ . Variabila  $m$  reține indicele de mijloc al subtabloului curent.

Dacă  $r-l=1$ , problema nu mai necesită o nouă descompunere, iar soluția reprezintă valoarea  $l$ .

2. **Rezolvarea subproblemelor obținute** prin descompunerea problemei curente. Această rezolvare este posibilă prin continuarea procesului recursiv de descompunere, până la obținerea unor subprobleme cu rezolvare imediată.

3. **Combinarea soluțiilor subproblemelor:** În cazul de față, această etapă nu există.

Implementarea strategiei descrise este realizată în cadrul funcției *SqrtDI()*:

```

1 var n:integer;
2
3 function SqrtDI(n,l,r:integer)
4   :integer;
5 var m:integer;
6 begin
7   if l = r-1 then SqrtDI := 1
8   else begin
9     m:=(l+r) div 2;
10    if m * m <= n then
11      SqrtDI := SqrtDI(n, m, r)
12    else
13      SqrtDI := SqrtDI(n, l, m)
14    end;
15  end;
16
17 begin
18  readln(n);
19  writeln(SqrtDI(n,0,n+1));
20 end.

```

#### 4. Sumă în arbore

Considerăm un arbore binar, memorat în heap. Realizați un subprogram, care returnează suma cheilor din arbore, folosind metoda *Divide et Impera*.

##### Soluție:

1. **Descompunerea problemei curente:** Suma cheilor unui arbore se poate descompune în două subprobleme: una care determină suma cheilor din subarborele stâng, iar cealaltă, în subarborele drept.

Dacă subarborele curent este vid, problema nu mai necesită o nouă descompunere, iar soluția o reprezintă valoarea 0.

2. **Rezolvarea subproblemelor obținute** prin descompunerea problemei curente. Această rezolvare este posibilă prin continuarea procesului recursiv de descompunere, până la obținerea unor subprobleme cu rezolvare imediată.

3. **Combinarea soluțiilor subproblemelor:** Suma cheilor unui arbore se obține însumând cheia rădăcinii cu soluțiile celor două subprobleme în care a fost descompusă problema curentă.

Luăm în considerație următoarele declarații:

```

type varf = ^nod;
nod = record
  inf : byte;
  fs, fd : varf;
end;

```

```

struct varf {
  int inf;
  varf *fs, *fd;
};

int sum(varf *p)
{
  if (p==NULL)
    return 0;
  return p->inf + sum(p->fs) +
    sum(p->fd);
}

```

Implementarea strategiei descrise, este realizată în cadrul funcției *Sum()*:

```

1 function Sum(p:varf):integer;
2 begin
3   if p=nil then Sum:=0
4   else
5     Sum := p^.inf +
6     Sum(p^.fs) + Sum(p^.fd)
7   end;

```

#### 5. Pătrat

Considerăm o valoare naturală  $n$  ( $n \leq 100$ ), de forma  $2^k$ . Să se construiască, în memorie, o matrice  $a$  patratică de ordin  $n$ , ale cărei elemente aparțin multimii {0,1}. În matrice, elementele egale cu 1 sunt cele situate în subtabloul stânga sus, de latură  $n \text{ div } 2$ . Procesul se repetă pentru cele trei subtablouri rămasă, până la obținerea unor subtablouri de latură 1. Realizați un subprogram care construiește matricea  $a$  după regula descrisă.

##### *Exemplu:*

Pentru  $n=4$

Matricea  $a$  va conține elementele

```

1 1 1 0
1 1 0 0
1 0 1 0
0 0 0 0

```

##### Soluție:

1. **Descompunerea problemei curente:** Identificăm tabloul curent, prin indicele elementului stânga sus ( $x, y$ ) și prin lungimea laturii sale  $l$ . Tabloul curent se împarte în 4 subtablouri de laturi  $l \text{ div } 2$ . Problema se descompune în trei subprobleme ce prelucrează trei dintre cele patru subtablouri.

Dacă lungimea laturii este 1, problema nu mai necesită o nouă descompunere.

2. **Rezolvarea subproblemelor obținute** prin descompunerea problemei curente. Această rezolvare este posibilă prin continuarea procesului recursiv de descompunere până la obținerea unor subprobleme cu rezolvare imediată.

3. **Combinarea soluțiilor subproblemelor:** În cazul problemei de față, această etapă nu mai este necesară.

Subprogramul *marc()* implementează strategia descrisă anterior:

```

1 procedure marc(x, y, l:byte);
2 begin
3   l:=l div 2;
4   for i:=x to x+l-1 do
5     for j:=y to y+l-1 do
6       a[i,j]:=1;
7   if l > 0 then begin
8     marc(x, y+1, l);
9     marc(x+1, y, l);
10    marc(x+1, y+1, l);
11  end;
12 end;
  
```

#### 6. Căutare binară

Se consideră un vector ce conține  $n$  numere întregi, ordonate strict crescător. Realizați o funcție, care primind o valoare întreagă, prin parametrul  $x$ , returnează poziția pe care aceasta se află în vector. În cazul în care căutarea se încheie fără succes, funcția va returna valoarea 0. Complexitatea algoritmului de căutare va fi  $O(\log_2 n)$ .

#### Soluție:

Cunoaștem că o căutare secvențială (element cu element) conduce la obținerea soluției problemei în complexitate liniară  $O(n)$  – incorrect având în vedere cerința. Pornind de la enunțul problemei, observăm că elementele sunt ordonate strict crescător. Pe această proprietate a tabloului se bazează strategia de căutare, folosind metoda *Divide et Impera*.

1. *Descompunerea problemei curente:* Identificăm tabloul curent, prin indicii limită  $(l, r)$ . Problema se descompune în două subprobleme ce prelucrează subtablourile delimitate de indicii  $(l, m - 1)$ , respectiv  $(m + 1, r)$ . Variabila  $m$  reține indicele de mijloc al tabloului inițial.

Dacă elementul  $A[m] = x$ , problema nu mai necesită o nouă descompunere, soluția fiind indicele  $m$ . În caz contrar, dacă valoarea  $x$  este mai mică decât elementul  $A[m]$ , atunci căutarea nu se va face decât în subtabloul delimitat de indicii  $(l, m - 1)$ , altfel, în subtabloul delimitat de indicii  $(m + 1, r)$ .

2. *Rezolvarea subproblemelor obținute:* Procesul va continua pentru una dintre cele două subprobleme obținute. Se observă că acest algoritm bazat pe strategia *Divide\_et\_impera* nu urmărește, în fond, o descompunere a problemei inițiale în subprobleme, ci o reducere a acesteia la o singură subproblemă de dimensiune mai mică. Cu alte cuvinte, din două subprobleme, se decide rezolvarea doar a uneia.

3. *Combinarea soluțiilor subproblemelor:* În cazul problemei de față, această etapă nu mai este necesară.

```

1 function CautB(l,x,r:integer)
2   :integer;
3 var m:integer;
4 begin
5   if l > r then CautB := -1
6   else
7     begin
8       m := (l + r) div 2;
9       if a[m] = x then CautB := m
10      else
11        if a[m] < x then
12          CautB := CautB(l, m-1, x)
13        else
14          CautB := CautB(m+1, r, x)
15      end;
16 end;
  
```

```

int cautB(int l,int x,int r)
{
  int m;
  if (l>r)
    return -1;
  m=(l+r)/2;
  if (a[m]==x)
    return m;
  if (a[m]>x)
    return cautB(l, m-1, x);
  return cautB(m+1, r, x);
}
  
```

#### 7. Sortarea prin interclasare - MergeSort

Se consideră un sir de  $n$  ( $n \leq 100$ ) numere naturale, memorate cu ajutorul unui vector  $A$ . Să se realizeze un subprogram care implementează algoritmul de sortare prin interclasare (*mergesort*).

#### Soluție:

Strategia de sortare prin interclasare utilizează metoda *Divide-et-Impera*.

1. *Descompunerea problemei curente:* Identificăm tabloul curent, prin indicii limită  $(l, r)$ . Problema se descompune în două subprobleme ce prelucrează subtablourile delimitate de indicii  $(l, m)$ , respectiv  $(m + 1, r)$ . Variabila  $m$  reține indicele de mijloc al tabloului inițial. Ambele probleme urmăresc sortarea tabloului pe care îl prelucrează.

Dacă  $l=r$ , problema nu mai necesită o nouă descompunere, tabloul fiind sortat.

2. *Rezolvarea subproblemelor obținute:* Rezolvarea problemei curente este posibilă prin continuarea procesului recursiv de descompunere, până la obținerea unor subprobleme cu rezolvare imediată. Procesul va continua pentru fiecare dintre cele două subprobleme obținute.

3. *Combinarea soluțiilor subproblemelor:* Operația de interclasare va constitui faza de combinare a soluțiilor subproblemelor obținute pe parcurs.

Sortarea prin interclasare are complexitatea  $O(n \log_2 n)$ . Subprogramul *interc()* realizează etapa de combinare a soluțiilor, interclasând subtablourile delimitate prin indicii  $(l, m)$  respectiv  $(m+1, r)$ .

```

1 procedure interc(l,m,r:byte);
2 var i,j,t,k:byte;
3   b:array[1..100] of byte;
4 begin
5   for i:=l to r do b[i]:=a[i];
6   i:=l;
7   j:=m+1;
8   k:=1;
9   while (i<=m) and (j<=r) do
10    if b[i]>b[j] then begin
11      a[k]:=b[j];
12      inc(j);
13      inc(k);
14    end
15   else begin
16     a[k]:=b[i];
17     inc(i);
18     inc(k);
19   end;
20   for t:=i to m do begin
21     a[k]:=b[t];
22     inc(k);
23   end;
24   for t:=j to r do begin
25     a[k]:=b[t];
26     inc(k);
27   end;
28 end;
29
30 procedure mergesort(l,r:byte);
31 var m:byte;
32 begin
33   if l <> r then begin
34     m:=(l+r) div 2;
35     mergesort(l, m);
36     mergesort(m+1, r);
37     interc(l, m, r);
38   end;
39 end;

```

### 8. Sortarea rapidă - QuickSort

Se consideră un sir de  $n$  ( $n \leq 100$ ) numere naturale memorate cu ajutorul unui vector  $A$ . Să se realizeze un subprogram care implementează algoritmul de sortare rapidă (quicksort).

#### Soluție:

Strategia de sortare prin interclasare utilizează metoda *Divide-et-Impera*.

*1. Descompunerea problemei curente:* Identificăm tabloul curent, prin indicii limită ( $l, r$ ). Ultimul element al tabloului ( $A[r]$ ), este plasat pe poziția corectă ( $poz$ ) în tabloul ordonat. În plus, toate elementele de valori mai mici decât  $A[r]$  se vor afla în stânga acestuia, iar cele mai mari, în dreapta lui.

```

void interc(int l,int m,int r)
{
  int i,j,t,k,b[101];
  for (i=l;i<=r;i++)
    b[i]=a[i];
  i=l;
  j=m+1;
  k=1;
  while (i<=m&&j<=r)
    if (b[i]>b[j])
      a[k++]=b[j++];
    else
      a[k++]=b[i++];
  for (t=i;t<=m;t++)
    a[k++]=b[t];
  for (t=j;t<=r;t++)
    a[k++]=b[t];
}

void mergesort(int l,int r)
{
  int m;
  if (l==r) return;
  m=(l+r)/2;
  mergesort(l, m);
  mergesort(m+1, r);
  interc(l, m, r);
}

```

Problema se descompune în două subprobleme care prelucrează subtablourile delimitate de indicii ( $l, poz - 1$ ), respectiv ( $poz + 1, r$ ). Ambele probleme urmăresc sortarea tabloului pe care îl prelucrează.

Dacă  $l = r$ , problema nu mai necesită o nouă descompunere.

*2. Rezolvarea subproblemelor obținute:* Rezolvarea problemei curente este posibilă prin continuarea procesului recursiv de descompunere, până la obținerea unor subprobleme cu rezolvare imediată. Procesul va continua pentru fiecare din cele două subprobleme obținute.

Sortarea rapidă are complexitatea  $O(n \log_2 n)$ . Subprogramul *Part()* realizează operația de plasare corectă a elementului  $A[r]$  și rearanjarea elementelor mai mici, respectiv, mai mari decât el. Funcția returnează indicele pe care a fost plasat.

```

1 function swap(var i,j:byte);
2 var t:integer;
3 begin
4   t:=i;
5   i:=j;
6   j:=t;
7 end;
8
9 function Part(l,r:byte):byte;
10 var p:integer;
11 begin
12   p := a[r];
13   j:=l-1;
14   for i:=l to r do
15     if a[i]<=p then
16       begin
17         inc(j);
18         swap(a[j],a[i]);
19       end;
20   Part := j;
21 end;
22
23 procedure Quicks(l,r:byte);
24 var poz:byte;
25 begin
26   poz := Part(l,r);
27   if l < poz - 1 then
28     Quicks(l,poz - 1);
29   if r > poz + 1 then
30     Quicks(poz + 1,r);
31 end;
32
33 void swap(int &i, int &j) {
34   int t;
35   t = i;
36   i = j;
37   j = t;
38 }
39
40 int part(int l,int r) {
41   int p,i,j;
42   p=a[r];
43   j=l-1;
44   for (i=l; i<=r; i++)
45     if (a[i]<=p)
46       swap(a[i+j], a[i]);
47   return j;
48 }
49
50 void quickS(int l,int r)
51 {
52   int poz;
53   poz=part(l,r);
54   if (l<poz-1)
55     quickS(l,poz-1);
56   if (r>poz+1)
57     quickS(poz+1,r);
58 }

```

## 9. Selectia celui de-al k-lea minim

Se consideră un sir de  $n$  ( $n \leq 100$ ) numere naturale memorate cu ajutorul unui vector  $A$ . Să se realizeze funcția *Sel* care returnează cel de-al  $k$ -lea element minim din vector. Funcția va schimba ordinea elementelor din  $A$ , astfel încât, la finalul execuției ei, pe poziția  $k$  în vector se va afla cel de-al  $k$ -lea număr cel mai mic.

### *Exemplus*

Pentru  $n=6$ ,  $k=4$  și vectorul  
 $a=(14, 23, 2, 1, 4, 21)$

| Funcția va returna valoarea 14, iar  
vectorul va conține (4, 1, 2, 14, 23, 21)

### Solutie:

Orice algoritm de sortare rezolvă problema de mai sus în  $O(n^2)$ . Se poate utiliza însă, metoda *Divide et Impera*, care va reduce complexitatea la  $O(n \log n)$ .

*1. Descompunerea problemei curente:* Identificăm tabloul curent, prin indicarea limită ( $l, r$ ). Ultimul element ( $A[r]$ ), este plasat pe poziția corectă ( $x$ ) în tabloul ordonat. În plus, toate elementele de valori mai mici decât  $A[r]$  se vor afla în stânga acestuia, iar cele mai mari, în dreapta lui.

Dacă poziția  $x = k$ , atunci problema este rezolvată, altfel problema este descompusă în două subprobleme, după cum urmează:

- Dacă  $x > k$ , se continuă cu subproblema care prelucrează prima parte a tabloului, între indicii  $(l, x - 1)$ ;
  - Dacă  $x < k$ , se continuă cu subproblema care prelucrează ultima parte a tabloului, între indicii  $(x + 1, r)$ .

*2. Combinarea soluțiilor subproblemelor:* În cazul problemei de față, nu este necesară.

Funcția *Part()* realizează operația de plasare corectă a elementului  $A[r]$  și reordonarea elementelor mai mici, respectiv mai mari decât el. Funcția returnează indicele pe care a fost plasat. Implementarea ei este prezentată anterior, în cadrul algoritmului de sortare rapidă.

```

1 function Sel(k,l,r:byte):byte
2 var x:byte;
3 begin
4   x := Part(l, r);
5   if x = k then Sel := a[k]
6   else
7     if x < k then
8       Sel := Sel(k, x + 1, r)
9     else
10      Sel := Sel(k, l, x - 1)
11 end;

```

```

int sel(int k,int l,int r)
{
    int x;
    x=part(l,r);
    if (x==k)
        return a[k];
    if (x<k)
        return sel(k,x+1,r);
    return sel(k,l,x-1);
}

```

## 10. Turnurile din Hano

Fie  $n$  ( $n \leq 10$ ) discuri de diametre diferite și trei tije identificate prin caracterele  $A$ ,  $B$  și  $C$ . Inițial, toate discurile sunt plasate pe tija  $A$ , în ordinea descrescătoare a diametrelor. Realizați un subprogram care identifică seria de mutări prin care toate discurile sunt mutate pe tija  $C$ , folosind ca tijă auxiliară tija  $B$ . La fiecare pas, se mută un singur disc care poate fi așezat doar peste un disc cu diametrul mai mare.

### Solutie:

### Descompunerea problemei curente

Problema mutării celor  $n$  discuri se descompune în trei probleme similare

- mutarea primelor  $n-1$  discuri de pe tija A pe tija B;
  - mutarea discului de pe tija A pe tija C;
  - mutarea celor  $n-1$  discuri de pe tija B pe tija C.

Cea de a doua subproblemă nu necesită alte descompuneri. Celelalte două subprobleme se descompun în mod asemănător, până la obținerea unor probleme de dimensiune 1.

Subprogramul *TH()* implementează algoritmul descris. Parametrii *a*, *b*, *c* rețin identificatorii tijei sursă, tijei destinație și tijei auxiliare (în această ordine).

```

1 procedure TH(n:byte;a,b,c:char)
2 begin
3   if n=1 then
4     writeln(a,' => ',b)
5   else begin
6     TH(n-1, a, c, b);
7     writeln(a,' => ',b);
8     TH(n-1, c, b, a);
9   end;
10 end;

```

```

void TH(int n,char a,char b,char c)
{
    if (n==1)
        printf("%c => %c\n",a,b);
    else {
        TH(n-1, a, c, b);
        printf("%c => %c\n", a, b);
        TH(n-1, c, b, a);
    }
}

```

### **3.2.2 Probleme propuse**

1. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente întregi. Să se realizeze un subprogram care determină suma și produsul elementelor (simultan), folosind metoda *Divide et Impera*.
  2. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente întregi. Să se realizeze un subprogram care determină cel mai mic și cel mai mare element (simultan), folosind metoda *Divide et Impera*.

3. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente întregi. Să se realizeze un subprogram care determină numărul de aparitii în tablou a unei valori citite de la tastatură, folosind metoda *Divide et Impera*.

4. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente întregi. Să se realizeze o funcție care determină al  $n$ -lea termen al sirului lui Fibonacci, folosind metoda *Divide et Impera*.

5. Se consideră o matrice pătratică  $n \times n$  ( $n \leq 100$ , de forma  $2^n$ ), cu elemente întregi. Să se realizeze un subprogram care afișează, folosind metoda *Divide et Impera*, pozițiile pe care se regăsește o valoare  $x$  citită de la tastatură.

6. În fișierul text *catalog.in*, pe prima linie, se află numărul  $n$  ( $n \leq 100$ ) de elevi ai unei clase, iar pe următoarele  $n$  linii se află scrise numele, prenumele și media anuală a fiecărui elev. Datele în cadrul unei linii sunt despărțite prin câte un spațiu. Să se realizeze un program care, citind de la tastatură numele și prenumele unui elev, afișează media anuală a acestuia. Programul va fi realizat cu ajutorul algoritmului de *căutare binară*.

7. Fiind dat un tablou bidimensional  $2^n \times 2^n$  cu elemente din mulțimea  $\{0,1\}$ , să se realizeze un program care identifică, folosind metoda *Divide et Impera*, cel mai mare subtablou ce conține un număr par de elemente egale cu 1. Subtabloul este obținut prin împărțiri succesive în patru zone de dimensiuni egale, începând cu tabloul initial și continuând cu fiecare dintre cele patru subtablouri, și.a.m.d. Se vor afișa indicii colțului stânga sus și dimensiunea laturii.

*Exemplu:*

Pentru  $n=8$  și tabloul:

0 1 0 1 0 0 0 0  
0 0 1 0 0 1 0 0  
0 0 0 0 0 0 0 0  
0 0 0 0 0 0 0 0  
0 0 1 0 0 1 0 0  
0 0 1 0 0 0 0 0  
1 0 1 0 0 0 0 0  
0 1 1 0 0 0 0 0

Se va afișa:

5 1  
4

8. Se consideră un tablou unidimensional de lungime  $n$  ( $n \leq 100$ ). Asupra acestuia se vor efectua operații de "tăiere" care constau în "înjumătățirea" lui și îndepărarea elementului din mijloc, dacă lungimea tabloului este impară. Procesul de tăiere se repetă, asupra fiecărei "jumătăți" până la obținerea unui tablou cu un singur element.

Să se afișeze, pe o singură linie, în fișierul text *out.txt*, toate elementele care rămân la încheierea procesului de tăiere.

Datele de intrare se citesc din fișierul *in.txt*, în următorul format: de pe prima linie, numărul  $n$ , iar de pe următoarea linie, numerele vectorului.

*Exemplu:*

*in.txt*

7

1 2 3 4 5 6 7

*out.txt*

1 3 5 7

9. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente întregi. Să se realizeze un program care determină câte numere din vector sunt mai mari decât un număr  $x$  dat, folosind un algoritm bazat pe metoda *Divide et Impera*.

*Exemplu:*

Pentru  $n=6$ ,  $x=7$  și tabloul  
(8, 3, 5, 1, 9, 4)

Se va afișa:  
2 elemente mai mari decât 7

10. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente întregi. Să se realizeze un program care determină câte elemente din vector conțin  $x$  cifre de 1 în scrierea binară, folosind un algoritm bazat pe metoda *Divide et Impera*.

*Exemplu:*

Pentru  $n=6$ ,  $x=1$  și tabloul  
(8, 3, 5, 1, 9, 4)

Se va afișa:  
3

11. Se cunosc greutățile a  $n$  ( $n \leq 100$ ) obiecte și greutatea maximă care poate fi transportată cu unicul rucsac avut la dispoziție. Să se realizeze un program care determină numărul maxim de obiecte ce pot fi transportate în rucsac (folosind algoritmul de sortare prin interclasare) și greutatea efectivă a lor.

*Exemplu:*

Pentru  $n=5$ ,  $G_{max}=10$  și tabloul  
(8, 3, 5, 1, 9)

Se va afișa:  
3

12. În fișierul text *catalog.in*, pe prima linie se află numărul  $n$  ( $n \leq 100$ ) de elevi ai unei clase, iar pe următoarele  $n$  linii se află scrise numele, prenumele și media anuală a fiecărui elev. Datele în cadrul unei linii sunt despărțite prin câte un spațiu. Să se realizeze un program care, folosind algoritmul de sortare rapidă, ordonează elevii descrescător după medii. În final, lista ordonată a elevilor se va afișa în fișierul *catalog.out*, în același format ca la intrare.

13. Se consideră un tablou unidimensional cu  $n$  ( $n \leq 100$ ) elemente naturale. Să se realizeze un program care determină o permutare a acestuia cu proprietatea că, pe fiecare indice par, este plasată o valoare mai mare decât elementele alăturate. Pentru ordonarea elementelor vectorului, folosiți metoda sortării prin interclasare.

*Exemplu:*

Pentru  $n=7$  și tabloul  
(2, 12, 4, 6, 0, 8, 10)

Se va afișa:  
0 12 4 8 6 10 2

14. Se consideră o matrice de dimensiuni  $3^n \times 3^n$ , cu elementele egale cu 0. Se împarte matricea în nouă subtablouri de aceleasi dimensiuni. Elementele zonei din mijloc devin egale cu 1. Se repetă procedeul de  $k$  ori ( $k \leq n$ ), pentru fiecare din cele 8 zone rămase de la fiecare pas. Să se realizeze un program care afișează matricea după  $k$  etape.

15. Presupunem că avem la dispoziție un caroaj format din  $2^n \times 2^n$  pătrățele colorate aleator în roșu și negru. Tabloul va fi împărțit după cele două axe de simetrie verticală și orizontală. Procesul se repetă asupra celor 4 subtablouri, până când se obține unul ale cărui pătrățele sunt toate de aceeași culoare. Realizați un program care determină numărul minim de plieri efectuate și culoarea elementelor din subtabloul soluție.

Datele de intrare se citesc din fișierul `input.txt`, care conține  $2^n$  linii. Pe fiecare linie se vor afla  $2^n$  caractere, 'R' sau 'N', reprezentând culorile fiecărui pătrățel de pe linia respectivă. Rezultatul va fi afișat pe ecran.

*Exemplu:*

`input.txt`

RRRN

RNRN

NRRR

NRRR

Se va afișa:  
1 Rosu

### 3.3. Metoda programării dinamice

#### 3.3.1 Probleme rezolvate

##### 1. Subșir crescător de lungime maximă

Se consideră un sir  $A$  ce conține  $n$  elemente întregi. Să se realizeze un program care determină un subșir crescător de lungime maximă al acestuia. Un subșir crescător al lui  $A$  se poate descrie astfel:  $A_{i_1} \leq A_{i_2} \leq \dots \leq A_{i_k}$  și  $1 \leq i_1 < i_2 < \dots < i_k \leq n$ .

*Exemplu:*

Pentru  $n=8$  și sirul 2 1 3 2 4 8 6 7

Se va afișa: 2 3 4 6 7

*Soluție:*

Orice problemă care este abordată prin metoda programării dinamice, trebuie să aibă o structură optimă, adică soluția optimă să poată fi construită pe baza soluțiilor optime ale subproblemelor sale. Pe de altă parte, subproblemele în care se descompune nu sunt independente, ci se suprapun. Toate soluțiile subproblemelor sunt, în general, salvate într-o structură de date de tip tablou.

##### Descompunerea problemei în subprobleme:

În cazul problemei de față, o subproblemă se referă la determinarea unui subșir crescător de lungime maximă, care începe cu elementul  $A_i$ ,  $1 \leq i \leq n$ .

##### Structuri de date utilizate:

- Tabloul  $A$  reține elementele sirului inițial;
- Tabloul  $L(N)$  va memora în elementul  $L[i]$  lungimea celui mai lung subșir crescător care începe cu  $A[i]$ .

##### Relațiile de recurență:

- $L[n]=1$ ;
- $L[i]=1 + \max\{L[j] \mid A[i] \leq A[j], i < j \leq n\}$ .

Lungimea celui mai lung subșir crescător care începe cu  $A[i]$  este mai mare decăt lungimea maximă a unui subșir care începe cu un element  $A[j]$ , mai mare decăt  $A[i]$ , situat „după” poziția  $j$ : ( $i < j \leq n$ ). Aceasta deoarece  $A[i]$  poate fi plasat în fața acestui subșir maximal, determinat anterior.

Pentru tabloul  $A=(2 \ 1 \ 3 \ 2 \ 4 \ 8 \ 6 \ 7)$ , vectorul  $L=(5, 5, 4, 4, 3, 1, 2, 1)$ .

Algoritmul are o complexitate patratică  $O(n^2)$ :

```

1 var n,i,Im,p:integer;
2 L,a:array[0..100] of integer;
3
4 procedure dinamica;
5 var i,j,max:integer;
6 begin
7   L[n]:=1;
8   for i:=n-1 downto 1 do begin
9     max:=0;
10    for j:=i+1 to n do
11      if (L[j]>max)and(a[i]<=a[j])
12        then max:=L[j];
13    L[i]:=max+1;
14    if Im<L[i] then Im:=L[i];
15  end;
16 end;
17
18 procedure drum();
19 var t:integer;
20 begin
21   p:=1;
22   t:=0;
23   repeat
24     while (L[p]<Im)or(a[t]>a[p])
25       do inc(p);
26     write(a[p], ' ');
27     t:=p;
28     dec(Im);
29   until Im=0;
30 end;
31
32 begin
33   read(n);
34   for i:=1 to n do read(a[i]);
35   dinamica();
36   drum();
37 end.

```

```

#include <stdio.h>
int n,i,Im,p,L[101],a[101];

void dinamica() {
  int i,j,max;
  L[n]=1;
  for (i=n-1;i>=1;i--) {
    max=0;
    for (j=i+1;j<=n;j++)
      if (L[j]>max)&&a[i]<=a[j])
        max=L[j];
    L[i]=max+1;
    if (Im<L[i]) Im=L[i];
  }
}

void drum() {
  int t;
  t=0; p=1;
  do {
    while (L[p]!=Im)&&(a[t]>a[p])
      p++;
    printf("%d ",a[p]);
    t=p;
    Im--;
  } while (Im);
}

int main() {
  scanf("%d",&n);
  for (i=1;i<=n;i++)
    scanf("%d", &a[i]);
  dinamica();
  drum();
  return 0;
}

```

## 2. Subsir maximal

Se consideră o valoare naturală  $x$  și un sir  $A$  crescător, ce conține  $n$  elemente naturale. Să se realizeze un program care determină un subsir de lungime maximă, în care diferența între oricare două elemente alăturate este mai mare sau egală cu  $x$ :  $(A_{i+1} - A_i) \geq x$ ,  $1 \leq i < n$ .

Exemplu:

Pentru  $n=6$ ,  $x=4$  și sirul  $5\ 7\ 9\ 10\ 14\ 15$  se va afișa  $5\ 9\ 14\ 5\ 10\ 15$  sau  $5\ 10\ 14$ .

Soluție:

Descompunerea problemei în subprobleme:

În cazul problemei de față o subproblemă se referă la determinarea unui subsir de lungime maximă care respectă condiția impusă și care se termină cu elementul  $A_i$ ,  $1 \leq i \leq n$ .

Structuri de date utilizate:

- Tabloul  $A$  reține elementele sirului inițial.
- Tabloul  $L(N)$  va memora în elementul  $L[i]$  lungimea celui mai lung subsir, al cărui ultim element este  $A[i]$ .

Relațiile de recurență:

- $L[n] = 1$ ;
- $L[i] = 1 + \max\{L[j] | A[i] - A[j] \geq x, i < j \leq n\}$ .

Lungimea celui mai lung subsir care se termină cu  $A[i]$  este mai mare cu 1 decât lungimea maximă a unui subsir care se termină cu un element  $A[j]$ , mai mic decât  $A[i]$  cu cel puțin valoarea  $x$ , situat „înaintea” poziției  $i$  ( $1 \leq j < i$ ). Aceasta deoarece  $A[i]$  poate fi plasat la finalul acestui subsir maximal, determinat anterior. Pentru tabloul  $A=(5\ 7\ 9\ 10\ 14\ 15)$  și  $x=4$ , vectorul  $L=(1\ 1\ 2\ 2\ 3\ 3)$ .

Algoritmul are o complexitate pătratică  $O(n^2)$ .

```

1 var n,i,Lm,x,p:integer;
2 L,a:array[0..100] of integer;
3
4 procedure dinamica;
5 var i,j,max:integer;
6 begin
7   L[1] := 1;
8   for i:=2 to n do begin
9     max := 0;
10    for j:=1 to i-1 do
11      if (L[j] > max)and
12        (a[j] + x <= a[i])
13      then max := L[j];
14    L[i] := max + 1;
15    if Lm < L[i] then begin
16      p := i; Lm := L[i]; end;
17  end;
18 end;
19
20 #include <stdio.h>
21 int n,i,Lm,p,x,L[101],a[101];
22
23 void dinamica() {
24   int i,j,max;
25   L[1]=1;
26   for (i=2;i<=n;i++) {
27     max=0;
28     for (j=1;j<i;j++)
29       if (L[j]>max&&a[j]+x<=a[i])
30         max=L[j];
31     L[i]=max+1;
32     if (Lm<L[i]) {
33       Lm=L[i];
34       p=i;
35     }
36   }
37 }
```

```

19 procedure
20 drum(lm,p,urm:integer);
21 var i,j:integer;
22 begin
23   if p=0 then exit;
24   if (L[p]=lm)and ((urm=-1) or
25     (a[p]+x<=a[urm])) then
26     begin
27       drum(lm-1,p-1,p);
28       write(a[p],' ');
29     end
30   else
31     if lm > 0 then
32       drum(lm,p-1,urm);
33   end;
34 begin
35   read(n,x);
36   for i:=1 to n do read(a[i]);
37   dinamica();
38   drum(lm,p,-1);
39 end.
40
41 
```

```

void drum(int lm,int p,int urm)
{
  if (!p) return;
  if ((L[p]==lm)&&(urm== -1)) {
    a[p]+x<=a[urm])) {
    drum(lm-1,p-1,p);
    printf("%d ", a[p]);
  }
  else if (lm > 0)
    drum(lm,p-1,urm);
}

int main()
{
  int i;
  scanf("%d %d",&n,&x);
  for (i=1;i<=n;i++)
    scanf("%d",&a[i]);
  dinamica();
  drum(lm,p,-1);
  return 0;
}

```

## 3. Jocul Domino

În acest joc se folosesc piese dreptunghihulare de aceleași dimensiuni. Făța unei piese este împărțită printr-o linie, în două părți marcate printr-un număr de puncte (0..6). Se consideră un sir de  $n$  piese de domino. El se consideră bine aranjat dacă, pentru oricare două piese așezate consecutiv, părțile lor alăturate sunt marcate fie cu același număr de puncte, fie suma acestora este egală cu 6. Un exemplu de sir de domino bine aranjat este (0,2),(2,5)(1,3).

Realizați un program care determină dacă se poate obține un sir bine ordonat, având voie să rotiți piesele, dar nu să le schimbați locul în cadrul sirului. Afisați fie sirul bine ordonat, fie mesajul ‘Imposibil’. În fișierul de intrare *domino.in* se găsește un număr par de cifre mai mici sau egale cu șase, două câte două valori reprezentând marcajele de pe o piesă.

Exemplu:

*domino.in*  
2 3 4 5 6 5 0 3

*domino.out*  
3 2 4 5 5 6 0 3

Soluție:

Descompunerea problemei în subprobleme:

În cazul problemei de față, o subproblemă se referă la determinarea unui subsir corect care se termină cu elementul  $A_i$ ,  $1 \leq i \leq n$ .

### Structuri de date utilizate:

- Matricea  $P[N][2]$  reține cele două marcaje ale pieselor din sir;
- Matricea  $A[N][2]$  reține succesiv soluțiile subproblemelor. Elementele matricei  $A$  aparțin mulțimii {True, False} cu semnificația:
  - $A[i,0]=$  True, dacă se poate obține un sir până în poziția  $i$  cu piesa  $i$  nerotită și False, în caz contrar;
  - $A[i,1]=$  True, dacă se poate obține un sir până în poziția  $i$  cu piesa  $i$  rotită și False, în caz contrar.
- Elementele matricei  $T[1..100][0..1]$  aparțin mulțimii {0, 1} cu semnificația:
  - $T[i,0]=$  0, dacă piesa  $i-1$  apare în sir nerotită și 1, în caz contrar (piesa  $i$  se consideră nerotită);
  - $T[i,1]=$  0, piesa  $i-1$  apare în sir nerotită și 1, în caz contrar (piesa  $i$  se consideră rotită).

### Relațiile de recurență:

Algoritmul descris în întregime în pseudocod verifică, pentru fiecare piesă  $i$  din sir dacă poate fi plasată în mod corect la finalul subșirului bine aranjat care se termină cu piesa  $i-1$ . Algoritmul are o complexitate liniară  $O(n)$ .

Aranjarea pieselor din sirul soluție (dacă există) se afișează prin intermediul tabloului  $T$ .

```

1 intreg n, i, j, p[100][1..2], t[100][0..1]; logic a[100][0..1];
2 a[1][0] ← true;
3 a[1][1] ← true;
4 pentru i ← 2, n executa
5   daca ((p[i][1]=p[i-1][2])or(p[i][1]+p[i-1][2]=6))and
6     (a[i-1][0]=true) atunci
7       a[i][0] ← true;
8       t[i][0] ← 0;           //i nerotita și (i-1) nerotita
9
10    daca ((p[i][1]=p[i-1][1])or(p[i][1]+p[i-1][1]=6))and
11      (a[i-1][1]=true) atunci
12        a[i][0] ← true;
13        t[i][0] ← 1;          //i nerotita și (i-1) rotita
14
15    daca ((p[i][2]=p[i-1][2])or(p[i][2]+p[i-1][2]=6))and
16      (a[i-1][0]=true) atunci
17        a[i][1] ← true;
18        t[i][1] ← 0;          //i rotita și (i-1) nerotita
19
20    daca ((p[i][2]=p[i-1][1])or(p[i][2]+p[i-1][1]=6))and
21      (a[i-1][1]=true) atunci
22        a[i][1] ← true;
23        t[i][1] ← 1;          //i rotita și (i-1) rotita
24
25

```

```

26   i ← n;
27   daca a[n][0]=true atunci j ← 0
28     altfel j ← 1;
29   cat timp i>0 executa
30     daca j=1 atunci scrie p[i][1], p[i][2]
31       altfel scrie p[i][2], p[i][1];
32     j ← t[i][j];
33     i ← i - 1;
34
35 stop.

```

### 4. Etalon

Se consideră o mulțime de  $N$  etaloane de greutăți cunoscute folosite pentru cântărirea cu ajutorul unui taler. Scrieți un program care determină o submulțime de etaloane care pot fi folosite pentru măsurarea unei greutăți  $k$  ( $k \leq 10000$ ). Fișierul *etalon.in* va conține, pe prima linie, numerele  $N$  ( $N \leq 100$ ) și  $k$ , iar pe a doua linie, cele  $N$  greutăți ale etaloanelor date. Submulțimea soluție se va afișa, pe prima linie, în fișierul *etalon.out*.

#### Exemplu:

*etalon.in*  
5 16  
2 4 5 3 7

*etalon.out*  
7 5 4

#### Soluție:

#### Descompunerea problemei în subprobleme:

În cazul problemei de față, o subproblemă se referă la determinarea tuturor greutăților  $x$ ,  $1 \leq x \leq k$ , care pot fi cântărite cu primele  $i$  etaloane  $1 \leq i \leq n$ .

### Structuri de date utilizate:

- Tabloul  $A(N)$  reține greutățile etaloanelor.
- Tabloul  $S(K)$  va memora succesiv soluțiile subproblemelor. Astfel, elementul  $S[x] =$  True, dacă se poate cântări greutatea  $x$  și False, în caz contrar.
- Tabloul  $T(K)$  va memora în elementul  $T[x]$  greutatea ultimului etalon folosit pentru a măsura greutatea  $x$ .

### Relațiile de recurență:

- $S[0] =$  True;
- $S[j+A[i]] =$  True, dacă  $\{ S[j] \text{ and not } S[j+A[i]] \}$ , unde  $j + a[i] \leq k$  și  $1 \leq i \leq n$ .

Cu alte cuvinte, utilizând etalonul  $A[i]$  se poate măsura greutatea  $j + A[i]$ , dacă există deja posibilitatea de a măsura greutatea  $j$  (fără a se folosi  $A[i]$ ). Se presupune că anterior, greutatea  $j + A[i]$  nu fusese obținută.

Pentru a evita folosirea unui etalon de mai multe ori, se va efectua parcurgerea vectorului  $S$  de la cea mai mare greutate măsurată anterior, către 0.

Implementarea subprogramului  $Subm$  ține seama de următoarele declarații:

```
var n,i,j,k:integer;
a,t:array[1..10000] of integer;
s:array[0..10000] of boolean; #include <stdio.h>
int n,i,j,k,a[10001],t[10001];
char s[10001];
```

Subprogramul  $drum()$  afișează etaloanele folosite pentru obținerea greutății  $k$ . Algoritmul prezentat are complexitatea  $O(n*k)$ .

```
1  ...
2  procedure Subm;
3  var mx:integer;
4  begin
5    s[0]:=true; t[0]:=0; mx:=0;
6    for i:=1 to n do begin
7      for j:=mx downto 0 do
8        if s[j] and not s[j+a[i]] and
9          (j+a[i]<=k)
10       then begin
11         s[j+a[i]]:=true;
12         t[j+a[i]]:=a[i];
13         if j+a[i]>mx then mx:=j+a[i]
14       end;
15     end;
16   end;
17
18  procedure drum(k:integer);
19  begin
20    if k<>0 then begin
21      write(t[k], ' ');
22      drum(k-t[k]);
23    end;
24  end;
25
26  begin
27    citire; Subm; drum(k);
28  end.
```

### 5. Cereri – (ONI'97)

La un depozit sosesc, pe rând,  $n$  ( $n \leq 100$ ) clienți care solicită fiecare o cantitate dată de vin. Butoiul din magazinul de deservire (considerat de capacitate nelimitată) este inițial gol. Administratorul depozitului are un algoritm propriu de deservire a clientilor: în funcție de ceea ce are în butoi, dar și de inspirația de moment, el poate să răspundă: "Vă ofer întreaga cantitate solicitată!" sau "Nu vă pot oferi acum cantitatea solicitată, poate altădată!". După ce pleacă clientul refuzat, administratorul se duce și aduce în butoiul din magazin exact cantitatea solicitată de clientul respectiv. Cunoscând cele  $n$  cantități cerute de clienți, să se determine un mod de a răspunde solicitărilor astfel încât, în final, cantitatea de vin vândută să fie maximă.

Din fișierul de intrare *cereri.in* se citește, de pe prima linie numărul  $n$ , iar de pe următoarea, cantitățile  $c_i$  ( $0 \leq c_i \leq 100$ ), exprimate în litri, cerute de clienți, în ordinea sosirii lor. În fișierul *cereri.out* se va scrie cantitatea vândută.

*Exemplu:*

*cereri.in*

6

8 14 9 4 6

*cereri.out*

13

Au fost serviziți clienții 4 și 5

*Soluție:*

*Descompunerea problemei în subprobleme:*

În cazul problemei de față, o subproblemă se referă la determinarea numărului maxim de litri de vin ce poate rămâne în butoi, pentru fiecare cantitate ce a fost vândută primilor  $i$  clienți,  $1 \leq i \leq n$ .

*Structuri de date utilizate:*

- Tabloul  $C(N)$  reține cererile clientilor, exprimate în litri.
- Tabloul  $S(10000)$  va memora succesiv soluțiile subproblemelor. Astfel elementul  $S[x] = y$ , semnifică faptul că la vânzarea a  $x$  litri, administratorul mai are maximum  $y$  litri în butoi.
- Variabila  $max$  va reține cantitatea maximă vândută.

*Relațiile de recurență:*

- $S[0] = 0$ ;
- $S[j + C[i]] = \max\{S[j + C[i]], S[j] - C[i], 0 \leq j \leq max\}$ , pentru  $1 \leq i \leq n$ .

Cu alte cuvinte, elementul  $S[j + C[i]]$  reține cea mai mare valoare dintre ce se găsește actual în acest butoi și ce rămâne după servirea clientului  $i$  din butoiul în care se găseau  $S[j]$  litri. Pentru a evita servirea unui client de mai multe ori, se va efectua parcurgerea vectorului  $S$  de la cea mai mare cantitate vândută anterior, către 0. Algoritmul prezentat are complexitatea  $O(n * max)$ .

```
1  procedure solve;
2  var max:integer;
3  begin
4    for i:=1 to 10000 do s[i]:=-1;
5    s[0] := 0; max := 0;
6    for i:=1 to n do begin
7      for j:=max downto 0 do
8        if (s[j] >= c[i]) and
9          (s[j+c[i]] < s[j]-c[i])
10       then begin
11         s[j + c[i]] := s[j]-c[i];
12         if j + c[i] > max then
13           max := j + c[i]
14       end;
15     s[0] := s[0] + c[i];
16   end;
17   writeln(max);
18 end;
```

```
void solve()
{
  int max;
  for (i=1;i<=10000;i++) s[i]=-1;
  s[0]=max=0;
  for (i=1; i<=n; i++)
  {
    for (j=max;j>=0;j--)
      if (s[j]>=c[i]&&
          s[j+c[i]]<s[j]-c[i]) {
        s[j+c[i]]=s[j]-c[i];
        if (j+c[i]>max) max=j+c[i];
      }
    s[0]+=c[i];
  }
  printf("%d\n", max);
}
```

## 6. Camionul

O persoană are la dispoziție un camion cu care poate transporta o greutate  $G_{max}$  și vrea să efectueze un singur transport în urma căruia să obțină un profit maxim. El are la dispoziție  $n$  obiecte, pentru fiecare dintre ele cunoscând greutatea și profitul adus în urma transportului lui. Realizați un program care identifică obiectele care vor fi transportate, în condițiile date, pentru a se obține un profit maxim.

### Soluție:

#### Descompunerea problemei în subprobleme:

În cazul problemei de față, o subproblemă se referă la determinarea profitului maxim pentru fiecare greutate posibilă considerând primele  $i$  camioane,  $1 \leq i \leq n$ .

#### Structuri de date utilizate:

- Tablourile  $G(N)$  și  $P(N)$  rețin greutatea respectiv profitul pentru fiecare camion.
- Tabloul  $S(10000)$  va memora succesiv soluțiile subproblemelor. Astfel elementul  $S[x] = y$ , semnifică faptul că la o greutate de  $x$  se poate obține profit maxim  $y$ .
- Varibila  $max$  va reține greutatea maximă.

#### Relațiile de recurență:

- $S[0] = 0$ ;
- $S[j+G[i]] = \max\{S[j+G[i]], S[j] + P[i], 0 \leq j \leq max\}$ , pentru  $1 \leq i \leq n$ .

Pentru a evita folosirea unui camion de mai multe ori, se va efectua parcurgerea vectorului  $S$  de la cea mai mare greutate obținută anterior, către 0.

Algoritmul prezentat are complexitatea  $O(n * max)$ .

```

1 procedure solve;
2 var max,rez,i,j:integer;
3 begin
4   s[0]:=0;
5   max:=0;
6   for i:=1 to n do begin
7     for j:=max downto 0 do
8       if s[j+g[i]]<=s[j]+p[i] then
9         begin
10           s[j+g[i]]:=s[j]+p[i];
11           if j+g[i]>max then
12             max:=j+g[i];
13           end;
14       end;
15     rez:=0;
16     for i=0;i<=gmax;i++
17       if (rez<s[i]) rez:=s[i];
18     printf("%d\n", rez);
19   end;
  
```

## 7. Etape

Într-o etapă a campionatului național de fotbal s-au desfășurat  $n$  ( $n \leq 100$ ) partide. Cunoscându-se scorurile tuturor partidelor, identificați o submulțime de meciuri în care gazdele au înscris de  $p$  ori, iar oaspeții de  $q$  ori. Din fișierul *etape.in* se citesc, de pe prima linie, numerele  $n$ ,  $p$  și  $q$ , iar de pe următoarele  $n$  linii, câte două numere  $x,y$ , având semnificația: gazdele au înscris de  $x$  ori, iar oaspeții de  $y$  ori. Scorurile meciurilor alese vor fi afișate pe câte o linie, în fișierul text *etape.out*.

#### Exemplu:

##### *etape.in*

5	6	4
4	2	1
1	2	2
4	1	1
2	1	2
3	1	2

##### *etape.out*

3	1
2	1
1	2

### Soluție:

#### Descompunerea problemei în subprobleme:

În cazul problemei de față, o subproblemă se referă la determinarea tuturor scorurilor cumulate  $(k, t)$ ,  $1 \leq k \leq p$ ,  $1 \leq t \leq q$ , care pot fi obținute din primele  $i$  partide,  $1 \leq i \leq n$ .

#### Structuri de date utilizate:

- Matricea  $A[N][2]$  reține scorurile celor  $n$  meciuri.
- Matricea  $B[N][2]$  va memora succesiv soluțiile subproblemelor. Astfel elementul  $B[i][j] = x$ , dacă există o submulțime formată din  $x$  meciuri, în care gazdele au înscris de  $i$  ori, iar oaspeții de  $j$  ori.
- Matricea  $T[N][2]$ , va memora în elementul  $T[i][j]$  numărul ultimului meci folosit pentru a obține scorul  $(i, j)$ .

#### Relațiile de recurență:

- $B[A[i][1], A[i][2]] = 1$ ;
- $B[j+A[i][1]][k+A[i][2]] = B[j][k] + 1$ , dacă  $B[j][k] \neq 0$ .

Cu alte cuvinte, luând în considerare meciul  $i$ , se poate obține o submulțime în care gazdele au înscris de  $j + A[i][1]$  ori, iar oaspeții de  $k + A[i][2]$  ori, dacă există anterior posibilitatea obținerii scorului  $j - k$  (fără meciul  $i$ ).

S-a optat pentru implementarea în pseudocod a algoritmului. Subprogramul recursiv *Trace()*, permite afișarea scorurilor meciurilor alese, bazându-se pe valorile reținute în tabloul  $T$ .

```

1  sintreg m, i, j, p, q, b[100][1..2], t[100][1..2], a[100][1..2];
2
3  subBUILD(b[], a[], t[], l, p, q) //construieste tablourile b si t
4  x ← a[1][1]; y ← a[1][2]; b[x][y] ← 1; t[x][y] ← 1;
5  pentru i ← 2, n execută
6    pentru j ← x, 0, -1 execută
7      pentru k ← y, 10, -1 execută
8        dacă (b[j][k]≠0) and (b[j+a[i][1]][k+a[i][2]] = 0) atunci
9          b[j+a[i][1]][k+a[i][2]] ← b[j][k]+1;
10         t[j+a[i][1]][k+a[i][2]] ← i;
11
12        dacă b[a[i][1]][a[i][2]] = 0 atunci
13          b[a[i][1]][a[i][2]] ← 1;
14          t[a[i][1]][a[i][2]] ← i;
15
16        x ← x + a[i][1];
17        y ← y + a[i][2];
18
19
20
21
22 TRACE(i, j); //subprogramul se va apela ptr valorile p si q
23   dacă t[i][j] ≠ 0 atunci
24     TRACE(i-a[t[i][j]][1], j-a[t[i][j]][2]);
25     scrie(a[t[i][j]][1], a[t[i][j]][2]);
26

```

### 8. Ariciul

Planul unei livezi de formă dreptunghiulară, cu dimensiunile  $n \times m$ , este format din zone pătrate cu latura 1. În fiecare zonă crește un pom. Din fiecare pom în zona respectivă pot cădea jos câteva mere. În zona stânga-sus se află un arici. Ariciul dorește să ajungă în zona dreapta-jos. În livadă există restricții de deplasare: ariciul se poate mișca din zona curentă în zona vecină din dreapta sau de jos. Elaborați un program care determină numărul maxim de mere pe care le poate strânge ariciul deplasându-se în zona dorită.

Planul livezii este redat prin tabloul  $A$  cu  $n$  linii și  $m$  coloane ( $2 < n, m < 100$ ). Elementul  $A[i,j]$  al acestui tablou indică numărul de mere căzute din pom în zona cu coordonatele  $(i,j)$ . Fișierul text *arici.in* conține, pe prima linie, numerele  $n, m$  separate prin spațiu. Fiecare din următoarele  $n$  linii conțin câte  $m$  numere separate prin spațiu. Nici o valoare de pe linie nu depășește 1000. Fișierul text *arici.out* conține o singură linie, pe care se scrie numărul maxim de mere strânse de arici.

*Exemplu:*

*arici.in*

```

3 3
0 4 1
0 1 1
1 0 1

```

*arici.out*

7

### Soluție:

*Descompunerea problemei în subprobleme:*

În cazul problemei de șață, o subproblemă se referă la determinarea numărului maxim de mere care pot fi culese pentru a ajunge la poziția  $(i, j)$ .

*Structuri de date utilizate:*

- Tabloul  $A(N,M)$  reține numărul de mere pentru fiecare zonă.
- Tabloul  $B(N,M)$  va memora succesiv soluțiile subproblemelor. Astfel, elementul  $B[i,j] = k$ , semnifică faptul că pentru a ajunge la poziția  $(i, j)$  se pot obține maximum  $k$  mere.

*Relațiile de recurență:*

- $B[1,1] = A[1,1]$ ;
- $B[1,j] = B[1,j-1] + A[1,j]$ ;
- $B[i,1] = B[i-1,1] + A[i,1]$ ;
- $B[i,j] = \max(B[i-1,j], B[i,j-1]) + A[i,j]$ .

Algoritmul prezentat are complexitatea  $O(n*m)$ .

```

1 procedure solve;
2 begin
3   a[1,1]:=b[1,1];
4   for j:=2 to m do
5     b[1,j]:=b[1,j-1]+a[1,j];
6   for j:=2 to n do
7     b[i,1]:=b[i-1,1]+a[i,1];
8   for i:=2 to n do
9     for j:=2 to m do
10       b[i,j]:=max(b[i-1,j], b[i,j-1])
11       + a[i,j];
12   writeln(b[n,m]);
13 end;

```

```

void solve() {
  b[1][1]=a[1][1];
  for (j=2;j<=m;j++)
    b[1][j]=b[1][j-1]+a[1][j];
  for (i=2;i<=n;i++)
    b[i][1]=b[i-1][1]+a[i][1];
  for (i=2;i<=n;i++)
    for (j=2;j<=m;j++)
      b[i][j]=max(b[i-1][j],b[i][j-1])
      + a[i][j];
  printf ("%d\n", b[n][m]);
}

```

### 9. Pătrat

Fie o matrice  $A$ , de dimensiuni  $N \times M$ , ale cărei elemente pot fi 0 sau 1. Numim pătrat o mulțime de elemente  $A[i,j]$  ce formează un subtablou cu laturile egale. Să se determine aria maximă a unui pătrat din matricea  $A$ .

Valorile  $N$  și  $M$  ( $1 \leq N, M \leq 200$ ) se vor citi de pe prima linie a fișierului *patrat.in*. Pe fiecare din următoarele  $N$  linii, se află câte  $M$  valori din mulțimea {0,1}, nesperate prin spații, reprezentând elementele matricei  $A$ .

În fișierul *patrat.out* se va afișa valoarea ariei corespunzătoare pătratului maximal.



2. Se consideră o mulțime de  $N$  obiecte de valori cunoscute folosite pentru premierea unor sportivi. Scrieți un program ce determină care este numărul maxim de valori ale unui premiu, care se pot obține folosind obiectele date.

Fișierul *premi.in* va conține, pe prima linie, numărul  $N$  ( $N \leq 200$ ), iar pe a doua linie, cele  $N$  valori ale obiectelor date (valori  $\leq 100$ ). Rezultatul se va afișa pe prima linie în fișierul *premi.out*.

*Exemplu:*

*premi.in*

2

6 3

*premi.out*

3

Deoarece un premiu poate avea valorile:  
3, 6 și 9

3. Gigel are o secvență de  $N$  ( $N \leq 200$ ) numere întregi din intervalul  $[-10.000, 10.000]$  și vrea să găsească un subșir de sumă maximă cu proprietatea că oricare două elemente ale subșirului nu sunt aflate pe poziții consecutive în secvență. În fișierul *secv.in* se va găsi numărul  $N$  și apoi  $N$  numere întregi, iar în fișierul *secv.out* suma subșirului cerut.

*Exemplu:*

*secv.in*

7

3 7 5 – 1 6 6 2

*secv.out*

16

4. Se consideră două șiruri de maximum 200 numere întregi. Se cere să se determine subșirul crescător de lungime maximă al șirului obținut prin intercalarea, în orice mod, a celor două șiruri. Din fișierul *interc.in* se vor citi de pe prima linie, lungimile celor două șiruri, iar de pe următoarele două linii, elementele primului șir, respectiv ale celui de al doilea șir. Rezultatul se va afișa, pe o singură linie, în fișierul *interc.out*.

*Exemplu:*

*interc.in*

6 5

2 1 0 5 3 7

4 3 6 1 8

*interc.out*

0 3 4 6 7 8

S-a obținut după intercalarea următoare:

2 1 0 5 3 4 3 6 1 7 8

5. Se consideră un șir de  $N$  ( $N \leq 200$ ) numere naturale. Să se determine un subșir de lungime maximă, în care fiecare valoare conține mai puține cifre de 1, în reprezentarea binară, decât elementul următor.

În fișierul *sir.in* se vor găsi numărul  $N$  și apoi  $N$  numere întregi, iar în fișierul *sir.out* subșirul obținut.

*Exemplu:*

*sir.in*

7

3 7 5 1 6 6 9 1 5

*sir.out*

16 9 15

6. Se consideră un șir de  $N$  ( $N \leq 200$ ) numere naturale de cel mult două cifre. Să se determine o submulțime de sumă 3 formată din număr minim de valori. În fișierul *subm.in* se vor găsi, pe prima linie, numerele  $N$  și 3, iar pe a doua linie,  $N$  numere naturale. În fișierul *subm.out* se va scrie submulțimea obținută.

*Exemplu:*

*subm.in*

4 7

3 2 1 4

*subm.out*

3 4

7. Fieind date un set de  $n$  valori distincte de timbre și limita superioară  $k$  a numărului de timbre care pot fi lipite pe un pliic, determinați cea mai mare secvență de valori consecutive de la 1 la  $M$  centi care se poate obține.

Datele de intrare se citesc din fișierul *timbre.in* ce conține, pe prima linie, numerele  $k$  ( $k \leq 200$ ) și  $n$  ( $n \leq 50$ ). Pe următoarea linie se găsesc cele  $n$  valori mai mici decât 1000. Datele de ieșire se vor scrie în fișierul *timbre.out* care va conține un singur număr reprezentând numărul  $M$  (maxim) de valori consecutive care se pot forma cu maximum  $k$  timbre de valori date.

*Exemplu:*

*timbre.in*

5 2

1 3

*timbre.out*

13

8. Se consideră secvența de numere prime  $P_1, P_2, \dots, P_n, \dots$ . Un număr este super prim dacă și el este prim și dacă numărul de ordine în șirul numerelor prime este un număr prim. De exemplu 3, este super prim (stă pe poziția a 2-a), dar 7 nu este (poziția a 4-a). Realizați un program care descompune un număr dat ca sumă de numere super prime. Dacă există mai multe posibilități, se va afișa cea cu număr minim de termeni. Numărul  $N < 10000$  și se va citi din fișierul *super.in*. Fișierul *super.out* va conține, pe o singură linie, termenii sumei separați prin câte un spațiu.

*Exemplu:* Pentru  $N=6$  fișierul *super.out* va conține 3 3.

9. Într-o cameră se află  $N < 51$  persoane. Fiecare persoană este născută într-o dintre cele  $Z < 366$  zile ale unui an. Determinați zilele de naștere ale fiecărei persoane, astfel încât în cameră să existe  $K$  perechi de persoane născute în aceeași zi. În fișierul *days.in* se află numerele întregi  $N, Z$  și  $K$  separate prin câte un spațiu. În fișierul *days.out*, veți afișa o singură linie, care conține  $N$  valori întregi, cuprinse între 1 și  $Z$ , reprezentând zilele de naștere ale celor  $N$  persoane, astfel încât în cameră să existe  $K$  perechi de persoane născute în aceeași zi. Dacă există mai multe soluții, puteți afișa oricare dintre ele. Dacă nu există nici o soluție, atunci afișați în fișier numai valoarea 0.

*Exemplu:* Dacă fișierul *days.in* conține 5 365 4, în *days.out* se va afișa 1 1 1 2 2.

10. Se consideră un triunghi ce conține  $n$  ( $n \leq 100$ ) linii, pe fiecare aflându-se valori de cel mult 2 cifre. Să se realizeze un program care determină un drum format din  $n$  valori de sumă maximă, care începe cu numărul de pe prima linie și se termină cu o valoare de pe ultima linie. Dacă numărul  $x$  se află pe drum, atunci următoarea valoare de pe drum este situată în triunghi pe linia următoare, fie sub  $x$ , fie la stânga sau la dreapta lui  $x$ . În fișierul *prob.in* se găseste triunghiul de numere, iar în fișierul *prob.out* se va scrie un sir de  $n$  valori reprezentând drumul determinat.

<i>prob.in</i>	<i>prob.out</i>
2	2 4 3 8
3 4	
3 2 5	
8 1 3 2	

11. Considerăm o tablă de șah de dimensiune  $n \times n$  ( $n \leq 10$ ). Să se determine un drum de lungime minimă prin care un cal aflat în poziția  $(x_i, y_i)$  se deplasează în  $(x_f, y_f)$ .

*Exemplu:*

Pentru  $n=5$ , poziția de început  $(1,2)$  și  
poziția finală  $(2,5)$

Se va afișa:  
 $(1,2)(3,3)(2,5)$

12. Realizați un program care determină numărul de valori naturale formate cu  $n$  ( $n \leq 10$ ) cifre din mulțimea  $\{1,2,3,4\}$ , astfel încât cifrele 1 și 2 să nu se afle pe poziții alăturate.

*Exemplu:* Pentru  $n=3$ , se va afișa 14.

13. Se consideră mulțimea formată din primele  $n$  ( $n \leq 30$ ) valori naturale. Să se determine numărul de modalități de a o partitura în două submulțimi de aceeași sumă.

*Exemplu:* Pentru  $n=7$ , se va afișa 4.

14. Se consideră un sir ordonat de numere reprezentate în baza 2. Sirul conține toate numerele formate din  $n$  cifre dintre care  $k$  cifre de 1. Cunoscând un număr  $x$  mai mic decât  $n$ , realizați un program care afișează al  $x$ -lea număr din sir, în ordinea dată de reprezentarea în baza 10 a numerelor. În fișierul *biti.in* se găsesc numerele  $n, k, x$ . Rezultatul va fi afișat în fișierul *biti.out*.

*Exemplu:* Dacă fișierul *biti.in* conține 5 3 19, în *biti.out* se va afișa 10100.

15. Se consideră un sir de  $n$  ( $n \leq 500$ ) intervale de numere naturale, identificate prin limitele  $[a_i, b_i]$ . Să se determine un subșir al său, în care oricare două sunt disjuncte și care conține un număr maxim de numere. În fișierul *intv.in* se va găsi, pe prima linie numărul  $N$ , iar pe următoarele  $N$  linii, limitele intervalelor. În fișierul *intv.out* se vor scrie, pe căte o linie, limitele intervalelor subșirului ales.

*Exemplu:* *subm.in*

3	<i>subm.out</i>
6 20	1 4
5 8	
1 4	6 20

## 3.4. Metoda Greedy

### 3.4.1 Probleme rezolvate

#### 1. Premiul

La o festivitate de premiere, dirigintele clasei are  $n$  ( $n \leq 1000$ ) obiecte de valori cunoscute mai mici decât 100 lei. Știind că elevului, care a obținut premiul I, îi va fi înmână  $m$  obiecte, realizați un program care identifică valoarea maximă a premiului I și care obiecte au fost alese.

*Exemplu:*

Pentru  $n=8, m=4$  și valorile  
3, 7, 8, 1, 6, 8, 9, 5

Se va afișa:

32  
8 7 8 9

*Solutie:*

Algoritmul de rezolvare a problemei se bazează pe metoda de programare Greedy. Această metodă ia în considerare construcția unei soluții optime, fără să construiască toate soluțiile posibile. Enunțul unei probleme care va fi abordată prin metoda Greedy poate fi generalizat, după cum urmează:

Considerându-se o mulțime  $A$  cu  $n$  elemente, se cere să se determine o submulțime a sa (*Sol*) care satisface anumite restricții.

Prima variantă folosește două funcții caracteristice: *Select* și *Correct*. *Select* este o funcție care are rolul de a alege următorul element din mulțimea  $A$  care să fie prelucrat. Funcția *Correct* verifică dacă un element poate fi adăugat soluției intermediare *Sol*<sup>(i)</sup>, astfel încât noua soluție *Sol*<sup>(i+1)</sup>, care s-ar obține, să fie o soluție validă. Prezentăm, în continuare, pseudocodul pentru această primă variantă greedy. Să considerăm că numărul de elemente al soluției optime *Sol* este  $m$  ( $m \leq n$ ).

```

1: GREEDY (A[N], Sol[M])
2: Sol ← φ
3: pentru i ← 1, n execută
4:   x ← Select(A)
5:   dacă Correct(Sol, x) atunci
6:     B ← x;
7:   fin
8:   Sol ← Sol ∪ B;
9: scrie Sol1, Sol2, ..., SolM;
10: stop.

```

Dacă funcția *Select* este bine concepută, atunci putem fi siguri că soluția *Sol* este o soluție optimă. În caz contrar, soluția *Sol* găsită va fi doar o soluție posibilă, nu și optimă. Ea se poate apropiă însă mai mult sau mai puțin de soluția optimă *Sol'*, în funcție de criteriul de selecție implementat. Din acest motiv, orice algoritm greedy trebuie însoțit de o demonstrare a corectitudinii.

A doua variantă de implementare diferă de prima prin faptul că face o etapă inițială de prelucrare a mulțimii  $A$ . Poate fi vorba de o sortare a elementelor mulțimii  $A$ , conform unui anumit criteriu, ales de programator. După sortare, elementele vor fi prelucrate direct în ordinea obținută. Prezentăm, în continuare, pseudocodul pentru această a doua variantă *greedy*.

```

1. GREEDY (A[N], Sol[M])
2. Sol ← ∅
3. Prelucreaza(A)
4. pentru i ← 1, n execută
5.   x ← A[i]
6.   dacă Corect(Sol, x) atunci
7.     B ← x;
8.   ┌─────────────────────────────────────────────────────────────────────────┐
9.   └─────────────────────────────────────────────────────────────────────────┘
10. scrie Sol1, Sol2, ..., SolM;
11. stop.

```

La a doua variantă, dificultatea funcției *Select* nu a dispărut, ci s-a transferat subprogramului *prelucrează*. Dacă prelucrarea mulțimii  $A$  este bine făcută, atunci se va ajunge, în mod sigur, la o soluție optimă. Altfel se va obține doar o soluție posibilă, mai mult sau mai puțin apropiată de optim.

În cazul problemei de față, se va utiliza a două variantă de implementare a metodei *greedy*. Vectorul  $A$ , care memorează valoarea premiilor, va fi sortat descrescător. Soluția va reține cele mai mari  $m$  valori din  $A$ , deci primele  $m$  elemente ale sale. Subprogramul *Solve*, care implementează algoritmul descris, ia în considerare următoarele declarații:

```

var a:array[1..10000] of integer; //include <stdio.h>
n,m:integer;                   int n,m,a[10001];

```

Algoritmul prezentat furnizează soluția optimă. În funcție de metoda de sortare aleasă, subprogramul poate avea complexitatea  $O(n^2)$  sau  $O(n \log_2 n)$ . În implementarea prezentată în continuare, s-a utilizat o metodă pătratică de ordonare.

```

1. procedure solve;
2. var i,j,s,x:integer;
3. begin
4.   s:=0;
5.   for i:=1 to n-1 do
6.     for j:=i+1 to n do
7.       if a[i]<a[j] then begin
8.         x:=a[i]; a[i]:=a[j];
9.         a[j]:=x;
10.      end;
11.   for i:=1 to m do begin
12.     s:=s+a[i];
13.     write(a[i], ' ');
14.   end;
15.   write(s);
16. end;

```

```

void solve() {
    int i,j,s=0,x;
    for (i=1;i<n;i++)
        for (j=i+1;j<=n;j++)
            if (a[i]<a[j]) {
                x=a[i];
                a[i]=a[j];
                a[j]=x;
            }
    for (i=1;i<=m;i++) {
        s+=a[i];
        printf("%d ",a[i]);
    }
    printf("\n%d\n",s);
}

```

## 2. Timp de așteptare

La o stație de benzină funcționează o singură pompă, la care urmează să fie serviti  $n$  ( $n \leq 1000$ ) clienți. Cunoscându-se timpul necesar servirii fiecărui client, realizăm un program care determină o ordine de servire a lor, astfel încât timpul mediu de așteptare să fie minim.

*Exemplu:*

Pentru  $n=5$  și timpii de servire:  
3, 1, 1, 4, 2

Se va afișa:

1 1 2 3 4  
timp de așteptare mediu 14/5

*Solutie:*

Timpul mediu de așteptare se calculează ca medie aritmetică a timpilor de așteptare a celor  $n$  clienți.

$$T_{\text{mediu}} = T_{\text{total}}/n = (T_1+T_2+\dots+T_n)/n.$$

Minimizarea timpului mediu de așteptare se reduce la minimizarea timpului total de așteptare. Întrucât timpul de așteptare al clientului  $i$  reprezintă suma timpilor de servire a clientilor aflați înaintea lui ( $1..i-1$ ), vom sorta crescător timpii de servire ai celor  $n$  clienți.

După cum se observă, algoritmul folosește a două variantă de implementare a metodei *greedy*. Notând cu  $A$  vectorul care memorează timpii de servire a celor  $n$  clienți, obținem următoarea formulă de calcul a timpului total de așteptare:

$$T_{\text{total}} = \sum(n-i)*A[i], 1 \leq i \leq n-1.$$

Algoritmul prezentat furnizează soluția optimă. În funcție de metoda de sortare aleasă, programul poate avea complexitatea  $O(n^2)$  sau  $O(n \log_2 n)$ . În implementarea prezentată în continuare, s-a utilizat o metodă pătratică de ordonare.

```

1. var n:integer;
2. a:array[1..1000] of integer;
3.
4. procedure load;
5. var i:integer;
6. begin
7.   readln(n);
8.   for i:=1 to n do read(a[i]);
9. end;
10.
11. procedure solve;
12. var x,i,j,T:integer;
13. begin
14.   for i:=1 to n-1 do
15.     for j:=i+1 to n do
16.       if a[i]>a[j] then begin
17.         x:=a[i];
18.         a[i]:=a[j];
19.         a[j]:=x;
20.       end;
21.   T:=0;

```

```

#include <stdio.h>

int n,a[1001];

void load() {
    int i;
    scanf("%d",&n);
    for (i=1;i<=n;i++)
        scanf("%d",&a[i]);
}

void solve() {
    int x,i,j,t=0;
    for (i=1;i<n;i++)
        for (j=i+1;j<=n;j++)
            if (a[i]>a[j]) {
                x=a[i];
                a[i]=a[j];
                a[j]=x;
            }
    T=0;
}

```

```

21 for i:=1 to n-1 do begin
22   write(a[i], ' ');
23   inc(T, (n-i)*a[i]);
24 end;
25 writeln(a[n], '/'; T, '/'; n);
26 end;
27
28 begin
29   load;
30   solve
31 end.

```

### 3. Modalitate de plată

Se consideră  $n$  ( $n \leq 1000$ ) tipuri de bancnote, de valori diferite, din fiecare existând un număr nelimitat de bucăți. Să se determine o modalitate de plată a valorii  $S$ , folosind un număr minim de bancnote.

*Exemplu:*

Pentru  $n=6$ ,  $S=100$  și bancnotele de valori: 3, 15, 1, 5, 2

Se va afișa:  
6 \* 15  
2 \* 5

*Soluție:*

Pentru rezolvarea problemei, vom utiliza o metodă euristică bazată pe tehnica *greedy*. Pentru minimizarea numărului de bancnote, vom încerca să plătim din sumă, cât putem de mult, cu bancnote de valori maxime.

În acest scop, vom ordona descrescător vectorul  $A$ , ce reține valorile bancnotelor. Fie  $S$  suma de plată curentă. Vom identifica elementul maxim  $A[i]$ , care este mai mic sau egal cu  $S$ . Numărul de bancnote folosite pentru plata sumei curente va fi  $[S/A[i]]$ .

Acest procedeu de alegere nu conduce neapărat la soluția optimă, iar în unele cazuri, nu identifică nici o soluție posibilă, deși aceasta există.

De exemplu, pentru  $S=19$  și bancnotele 3, 5, 2 o soluție optimă ar fi 3 bancnote de valoare 3 și 2 bancnote de valoare 5. Algoritmul propus nu identifică nici o soluție, deoarece va alege să plătească cu 3 bancnote de valoare 5, o bancnotă de valoare 3, iar suma rămasă  $S=1$  nu poate fi plătită.

Implementarea algoritmului euristic propus are complexitatea  $O(n^2)$  deoarece s-a utilizat o metodă pătratică de ordonare.

```

1 var n,s:integer;
2 a:array[1..1000] of integer;
3
4 procedure load;
5 var i:integer;
6 begin
7   readln(n,s);
8   for i:=1 to n do read(a[i]);
9 end;

```

```

#include <stdio.h>
int n,s,a[1001];
void load() {
    int i;
    scanf("%d %d",&n,&s);
    for (i=1;i<=n;i++)
        scanf("%d",&a[i]);
}

```

```

10 procedure solve;
11 var x,i,j:integer;
12 begin
13   for i:=1 to n-1 do
14     for j:=i+1 to n do
15       if a[i]<=a[j] then begin
16         x:=a[i]; a[i]:=a[j]; a[j]:=x
17       end;
18   for i:=1 to n do
19     if a[i]<=S then begin
20       write(S div a[i],'*',a[i]);
21       S:=S - S div a[i] * a[i];
22       if S = 0 then break
23     end;
24   if S>0 then
25     writeln('raspuns eronat')
26 end;
27
28 begin
29   load;
30   solve
31 end.

```

```

void solve() {
    int x,i,j;
    for (i=1;i<n;i++)
        for (j=i+1;j<=n;j++)
            if (a[i]<=a[j]) {
                x=a[i]; a[i]=a[j];
                a[j]=x;
            }
    for (i=1;i<=n;i++)
        if (a[i]<=s) {
            printf("%d * %d\n",s/a[i],a[i]);
            s=(s/a[i])*a[i];
            if (!s) break;
        }
    if (s>0) printf("raspuns eronat");
}
int main() {
    load();
    solve();
    return 0;
}

```

### 4. Rucsac-varianța discretă (0/1)

Se consideră un rucsac cu care se poate transporta o greutate maximă  $G_{max}$  și mai multe obiecte de greutăți  $g_1, g_2, \dots, g_n$ , la transportul cărora se obțin câștigurile  $c_1, c_2, \dots, c_n$ . Se cere să se încarce rucsacul astfel încât să se obțină un câștig maxim. Rezultatul va fi afișat pe ecran printr-un sir de  $n$  cifre de 0 sau 1, având semnificația: obiectul  $i$  este pus în rucsac, dacă cifra  $i$  din sirul afișat este egală cu 1 și 0, în caz contrar.

*Exemplu:*

Pentru  $n=4$ ,  $G_{max}=10$   
 $g=(3, 2, 8, 1)$   
 $c=(6, 8, 8, 5)$

Se va afișa:  
1 1 0 1 (obiectele alese)  
19 (câștigul maxim)

*Soluție:*

Se observă că cea mai bună metodă de încărcare a rucsacului ar fi să introducem obiectele în ordinea descrescătoare a eficienței acestora. Vom calcula eficiența fiecărui obiect  $e_1=g_1/c_1$ ,  $e_2=g_2/c_2, \dots, e_n=g_n/c_n$  și vom sorta obiectele în ordinea descrescătoare a eficienței. Acestea fiind realizate aplicăm metoda Greedy:

- Inițial, greutatea obiectelor transportate este  $G=0$ ;
- Se alege un obiect în ordinea descrescătoare a eficienței;
- Verificăm dacă putem adăuga obiectul, adică dacă prin adăugare nu se depășește greutatea admisă;
- Repetăm procedeul până când s-au terminat obiectele sau până când nu mai există obiect care poate fi transportat.

În cadrul implementării algoritmului descris, s-a utilizat o înregistrare pentru memorarea atributelor fiecărui obiect : greutate ( $g$ ), câștig ( $c$ ), numărul de ordine al obiectului ( $nr$ ) și starea obiectului ( $x=0..1$ )—ales sau nu să fie transportat. Algoritmuluristic propus are complexitatea  $O(n^2)$ . Un algoritm care furnizează întotdeauna soluția optimă utilizează programarea dinamică.

```

1 type ob=record
2   g,c,nr:integer; x:0..1; end;
3 var a:array[1..1000] of ob;
4   n,m,gm:integer;
5
6 procedure load;
7 var i:integer;
8 begin
9  readln(n,gm);
10 for i:=1 to n do read(a[i].g);
11 for i:=1 to n do read(a[i].c);
12 for i:=1 to n do a[i].nr:=i;
13 end;
14
15 procedure solve;
16 var i,j,g,c:integer;
17 x:ob;
18 begin
19  g:=0;
20  c:=0;
21  for i:=1 to n-1 do
22    for j:=i+1 to n do
23      if a[i].c/a[i].g<
24        a[j].c/a[j].g
25      then begin
26        x:=a[i];
27        a[i]:=a[j];
28        a[j]:=x;
29      end;
30  for i:=1 to n do
31    if g+a[i].g<=gm then begin
32      inc(g,a[i].g);
33      inc(c,a[i].c);
34      a[i].x:=1;
35    end
36    else a[i].x:=0;
37  writeln(c);
38  for i:=1 to n do
39    for j:=1 to n do
40      if a[j].nr=i then
41        write(a[j].x,' ');
42  end;
43
44 begin
45  load;
46  solve;
47 end.

```

```

#include <stdio.h>

struct ob {
  int g,c,nr,x;
} a[1001];
int n,m,gm;

void load() {
  int i;
  scanf("%d %d",&n,&gm);
  for (i=1;i<n;i++) scanf("%d",&a[i].g);
  for (i=1;i<n;i++) scanf("%d",&a[i].c);
  for (i=1;i<n;i++) a[i].nr=i;
}

void solve() {
  int i,j,g=0,c=0;
  ob x;
  for (i=1;i<n;i++) {
    for (j=i+1;j<=n;j++) {
      if (a[i].g*a[j].c>a[i].c*a[j].g) {
        x=a[i];
        a[i]=a[j];
        a[j]=x;
      }
      for (i=1;i<n;i++) {
        if (g+a[i].g<=gm) {
          g+=a[i].g;
          c+=a[i].c;
          a[i].x=1;
        } else a[i].x=0;
      }
      printf("%d\n",c);
      for (i=1;i<n;i++)
        for (j=i+1;j<=n;j++) {
          if (a[j].nr==i)
            printf("%d ",a[j].x);
        }
    }
  }
}

int main() {
  load();
  solve();
  return 0;
}

```

### 5. Rucsac-varianța continuă

Se consideră un rucsac cu care se poate transporta o greutate maximă  $G_{max}$  și mai multe obiecte de greutăți  $g_1, g_2, \dots, g_n$ . În transportul cărora se obțin câștigurile  $c_1, c_2, \dots, c_n$ . Se cere să se încarce rucsacul astfel încât să se obțină un câștig maxim, știind că obiectele pot fi "împăiate".

Rezultatul va fi afișat pe ecran printr-un sir de  $n$  numere reale  $x_i$  cuprinse între 0 și 1, având semnificația procentului transportat din fiecare obiect.

#### Exemplu:

Pentru  $n=3$ ,  $G_{max}=8$   
 $g=(4,6,2)$   
 $c=(4,3,1)$

Se va afișa:  
 1 0.67 0  
 6 (câștigul maxim)

#### Soluție:

Ca și la problema anterioară, cea mai bună metodă de încărcare a rucsacului ar fi să introducem obiectele în ordinea descrescătoare a eficienței acestora. Vom calcula eficiența fiecărui obiect  $e_i=g_i/c_i$ ,  $e_2=g_2/c_2, \dots, e_n=g_n/c_n$  și vom sorta obiectele în ordinea descrescătoare a eficienței. Acestea fiind realizate aplicăm metoda Greedy:

- Inițial, greutatea obiectelor transportate este  $G=0$ .
- Se alege un obiect în ordinea descrescătoare a eficienței.
- Verificăm dacă putem adăuga obiectul în întregime, adică dacă prin adăugare nu se depășește greutatea admisă. În caz contrar, vom tăia obiectul păstrând greutatea permisă la transport.
- Repetăm procedeul până când s-au terminat obiectele sau până s-a încărcat greutatea  $G_{max}$  admisă.

Implementarea subprogramului *Solve* ia în considerare următoarele declarații:

```

type ob=record
  g,nr:integer; c,x:real end;
var a:array[1..1000] of ob;
  n,m,gm:integer;

```

```

#include <stdio.h>
struct ob {
  int g,nr; double c,x;
} a[1001]; int n,m,gm;

```

În cadrul implementării algoritmului descris, s-a utilizat o înregistrare pentru memorarea atributelor fiecărui obiect : greutate ( $g$ ), câștig ( $c$ ), numărul de ordine al obiectului ( $nr$ ) și procentul ales din obiect pentru a fi transportat ( $x=0..1$ ). Algoritmul propus, are complexitatea  $O(n^2)$ .

```

1 procedure solve;
2 var i,j:integer;
3   x:ob;c:real;
4 begin
5  for i:=1 to n-1 do
6    for j:=i+1 to n do
7      if a[i].c/a[i].g<
8        a[j].c/a[j].g
9      then begin
10        x:=a[i];a[i]:=a[j];a[j]:=x;
11      end;

```

```

void solve()
{
  int i,j;
  double c;ob x;
  for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
      if (a[i].c/a[i].g<
          a[j].c/a[j].g)
        (x=a[i]); a[i]=a[j];
        a[j]=x;
}

```

```

21 c := 0;
22 i := 1;
23 while (gm>0) and (i<=n) do
24 if a[i].g<=gm then begin
25 dec(gm,a[i].g);
26 c := c+a[i].c;
27 a[i].x := 1;
28 inc(i);
29 end
30 else begin
31 a[i].x := gm/a[i].g;
32 c := c + a[i].c*a[i].x;
33 gm := 0;
34 for j := i+1 to n do
35 a[j].x := 0;
36 i := n+1;
37 end;
38 writeln(c:0:2);
39 for i:=1 to n do
40 for j:=1 to n do
41 if a[j].nr=i then
42 write(a[j].x:0:2,' ')
43 end;

```

#### 6. Saci cu bani

La monetăria statului, sunt aduși  $n$  ( $n \leq 1000$ ) saci cu monede. Guvernatorul cunoaște câte monede sunt în fiecare sac și dorește ca toate să conțină același număr de monede. Pentru aceasta, el va efectua mutări de monede dintr-un sac în altul. Realizați un program care determină șirul minim de mutări care trebuie efectuate pentru a egală numărul de monede din fiecare sac. Din fișierul *saci.in*, se va citi, de pe prima linie, numărul  $n$ , iar de pe următoarea linie, numărul de monede din fiecare sac (valori mai mici ca 100000). În fișierul *saci.out* se va scrie pe fiecare linie câte un triplet de numere  $x$ ,  $y$ ,  $z$  având semnificația:  $z$  monede vor fi mutate din sacul  $x$  în sacul  $y$ .

*Exemplu:*

*saci.in*

3

24 17 19

*saci.out*

4 1 2

1 2 3

#### Solutie:

Presupunem că numărul total de monede din saci este divizibil cu  $n$ , în caz contrar, problema nu admite soluție. Notăm cu  $med$  numărul de monede care ar trebui să existe în fiecare sac la finalul algoritmului.

Procedeul *Greedy* propus identifică, la fiecare pas, sacul ce conține cele mai multe monede și pe cel care conține cele mai puține. Se va transfera surplusul din sacul cu mai multe monede în sacul cu cele mai puține monede. Procedeul continuă, cât timp numărul minim și maxim de monede din saci diferă de valoarea medie  $med$ .

Algoritmul are complexitate pătratică și nu garantează identificarea soluției optime.

```

1 var n,s,med:integer;
2 a:array[1..10000] of integer;
3
4 procedure citire;
5 var i:integer;
6 begin
7   readln(n);
8   s:=0;
9   for i:=1 to n do begin
10     read(a[i]); s:=s+a[i]; end;
11   med:=s div n;
12 end;
13
14 procedure rezolva;
15 var min,max,i,p1,p2:integer;
16 begin
17   min := s + 1; max := 0;
18   while(min>med) and
19     (max>med) do
20   begin
21     min := s+1; max := 0;
22     for i:=1 to n do begin
23       if min>a[i] then begin
24         min:=a[i]; p1:=i end;
25       if max<a[i] then begin
26         max:=a[i]; p2:=i end;
27     end;
28     a[p1]:= a[p1] + a[p2] - med;
29     if a[p2]-med <>0 then
30       writeln(a[p2]-med,' ',p2,' ',p1);
31     a[p2]:=med;
32   end;
33 end;
34
35 begin
36   citire;
37   rezolva;
38 end.

```

#### 7. Festivalul de film

Considerăm că la un festival de film, ce se desfășoară în sălile unui cinematograf, sunt proiectate într-o zi  $n$  filme (numerotate de la 1 la  $n$ ). Un cinefil dorește să urmărească, în întregime, cât mai multe filme. Evident, nu poate viziona două filme care se proiectează simultan. Cunoscând intervalul orar când este proiectat fiecare film, să se realizeze un program care identifică numărul maxim de filme care pot fi vizionate de cinefil și care dintre ele vor fi alese.

Din fișierul *film.in* se va citi, de pe prima linie, numărul  $n$  ( $n \leq 1000$ ), iar de pe următoarele linii, perechi de numere reale, semnificând ora de începere a filmelor și cea de final, de la primul la ultimul.

Exemplu:

film.in

5  
9.30 10.30  
8.20 10.10  
12.50 13.50  
7.00 13.30  
10.20 12.40

film.out

3  
253 (nr. de ordine al filmelor)

Soluție:

Pentru identificarea soluției optime, vom ordona filmele, crescător după ora de final. Inițial, selectăm în soluție filmul care se termină cel mai devreme. La fiecare pas vom adăuga la soluție, filmul care începe primul după ce se termină ultimul film selectat. Procedeul asigură identificarea corectă a soluției optime.

În cadrul implementării algoritmului descris, s-a utilizat o înregistrare pentru memorarea atributelor fiecărui film: ora de început (*s*), ora de final (*f*) și numărul de ordine al filmului (*nr*). Algoritmul propus, are complexitatea  $O(n^2)$ .

```

1 type sp=record
2   s,f:real; nr:integer; end;
3 var a:array[1..1000] of sp;
4   n,m:integer;
5
6 procedure load;
7 var i:integer;
8 begin
9 assign(input,'film.in');
10 reset(input); readln(n);
11 for i:=1 to n do begin
12   read(a[i].s,a[i].f);
13   a[i].nr:=i; end;
14 end;
15
16 procedure solve;
17 var u,i,j,nr:integer; x:sp;
18 begin
19   for i:=1 to n-1 do
20     for j:=i+1 to n do
21       if a[i].f>a[j].f then begin
22         x:=a[i]; a[i]:=a[j]; a[j]:=x;
23       end;
24   u:=1; nr:=1;
25   writeln(a[u].nr,'');
26   for i:=2 to n do
27     if a[u].f<=a[i].s then
28       begin
29         writeln(a[i].nr,''); u:=i;
30         inc(nr)
31       end;
32   writeln(nr);
33 end;
34
35 begin load; solve; end.

```

### 3.8. Minimizarea numărului de transporturi

Se consideră un număr neînlimitat de rucsaci de același tip, cu fiecare dintre ei putându-se transporta o greutate maximă  $Gm$  și  $n$  ( $n \leq 1000$ ) obiecte, de greutăți  $g_1, g_2, \dots, g_n$ . Realizați un program care determină numărul minim de rucsaci necesari pentru a transporta toate obiectele.

Exemplu:

Pentru  $n=5$ ,  $Gmax=10$   
 $g=(3,2,8,4,5)$

Se va afișa:  
3

Soluție:

Algoritmul heuristic propus caută, de la stânga la dreapta, pentru fiecare obiect, primul rucsac în care poate fi plasat. Observăm că, la sfârșitul algoritmului, nu pot exista doi rucsaci folosiți pe mai puțin de jumătate amândoi, pentru că atunci conținutul celui de-al doilea ar fi fost transferat în primul. Cu alte cuvinte, dacă însumăm, pentru toți rucsacii greutatea rămasă nefolosită, vom observa că aceasta este mai mică decât cea utilizată.

Tabloul *g* reține greutățile obiectelor, iar vectorul *l* memorează, pentru fiecare rucsac, greutatea suplimentară pe care o mai poate transporta. Inițial elementele vectorului *l* sunt egale cu greutatea *Gm* care poate fi transportată cu un rucsac. Variabila *use* reține numărul maxim de rucsaci folosiți.

```

1 #include <stdio.h>
2 int n,gm,g[1001],l[1001];
3
4 procedure load;
5 var i:integer;
6 begin
7   readln(n,gm);
8   for i:=1 to n do read(g[i]);
9 end;
10
11 procedure solve;
12 var i,j,use:integer;
13 begin
14   use := 0;
15   for i:=1 to n do l[i] := gm;
16   for i:=1 to n do begin
17     j := 1;
18     while l[j] < g[i] do inc(j);
19     if use < j then use := j;
20     dec(l[j], g[i]);
21   end;
22   writeln(use);
23 end;
24
25 begin
26   load; solve
27 end.

```

## 9. Expresie

Fie sirurile  $a[1], a[2], \dots, a[n]$  și  $b[1], b[2], \dots, b[m]$ ,  $m > n$ . Să se maximizeze valoarea expresiei  $E = a[1]x[1] + a[2]x[2] + \dots + a[n]x[n]$ , unde  $x[i]$  sunt elemente ale sirului  $b$ .

Datele de intrare se vor citi din fișierul `valmax.in` în formatul următor: pe prima linie, numerele  $n$  și  $m$ , iar pe următoarele două linii, elementele celor două siruri. Rezultatul va fi afișat pe ecran.

### Exemplu:

Pentru  $n=6$ ,  $m=8$  și sirurile  $a=(3, 7, -10, 5, -1, 2)$  respectiv  $b=(10, 5, 20, -20, -2, 7, 9, -10)$ , valoarea maximă a lui  $E$  este 441.

### Solutie:

Algoritmul are la bază principiul următor: pentru două siruri cu  $n$  elemente, respectiv  $m$  ( $n < m$ ) sortate crescător, dacă primul are  $k$  elemente negative și  $p$  elemente pozitive ( $k+p=n$ ), se cuplază primele  $k$  elemente din primul sir cu primele  $k$  din al doilea și ultimele  $p$  elemente din primul sir cu ultimele  $p$  din al doilea sir.

```

1 intreg n, m, a[100], b[100], i, j, r;
2 daca n>m atunci inverseaza a cu b;
3 sorteaza sirul a crescator;
4 sorteaza sirul b crescator;
5 r ← 0; i ← 0;
6 cat timp (a[i + 1] < 0)and(i + 1 <= n) executa
7     i ← i + 1;
8
9     pentru j ← 1, i executa
10        r ← r + a[j]*b[j];
11
12    pentru j ← i + 1, n executa
13        r ← r + a[j] * b[m - n + j];
14
15 scrie r;
16 stop.

```

### 3.4.2 Probleme propuse

- Se consideră un sir de  $n$  ( $n \leq 10000$ ) valori reale. Să se realizeze un program care determină o submulțime a acestuia, care are proprietatea că suma elementelor este maximă.
- Se consideră un sir de  $n$  ( $n \leq 1000$ ) valori naturale, reprezentând valoarea unor obiecte. Partiționați acest sir în două submulțimi care au proprietatea că diferența absolută a sumelor elementelor lor este minimă.

*Exemplu:* Pentru  $n=4$  și valorile 3, 19, 17, 1 se va afișa: 3, 17 și 1, 19.

- Se consideră  $n$  ( $n \leq 1000$ ) lucrări numerotate de la 1 la  $n$ , ce se pot executa fiecare într-o singură oră. Pentru fiecare lucrare, se cunosc ora când trebuie predată (valori naturale cuprinse între 1 și  $n$ ) și penalizarea pentru nerespectarea acesteia. Știind că nu pot fi executate două lucrări simultan, să se realizeze un program care identifică o ordine de executare a lucrărilor, având penalizare totală minimă.

*Exemplu:* Pentru  $n=4$  și orele de predare 2, 1, 1, 3 și penalizările 100, 20, 10, 10 se va afișa: 2 1 4 3 cu penalizare totală 10.

- Într-un hotel cu  $k$  ( $k \leq 100$ ) camere s-au primit pentru anul următor  $n$  ( $n \leq 1000$ ) solicitați de cazare, fiecare pentru  $m$  ( $m \leq 100$ ) zile. Pentru fiecare cerere se cunoaște ziua de început a perioadei de cazare. Știind că anul următor are 365 de zile, să se realizeze un program care determină numărul maxim de cereri ce pot fi satisfăcute.

- Se dau două siruri de lungime  $N$  și un număr  $K$  ( $N < 1000$ ,  $K < 1000$ ). Cele două siruri au numai numere 1 și -1. Scopul e să-l transformăm pe primul în al doilea. Singura operație permisă e să selectăm o secvență de  $K$  elemente alăturate și să le inversăm semnul la toate numerele cuprinse în această zonă. Datele vor fi citite de la tastatură. Fișierul `siruri.out` va conține, pe prima linie  $M$ , numărul minim de operații, iar, pe următoarele  $M$  linii, poziția de început de unde se aplică operația.

*Exemplu :* Pentru  $n=8$ ,  $k=2$  și sirul

1 -1 1 -1 -1 1 1

3
1
2
3

`siruri.out`

- Se consideră un număr  $n$  ( $n \leq 100$ ) și  $k \leq n$ . Să se construiască o matrice pătratică de ordin  $n$ , ale cărei elemente satisfac simultan condițiile:

- conține toate numerele de la 1 la  $n^2$ ;
- pe fiecare linie numerele sunt ordonate crescător;
- suma elementelor pe coloana  $k$  este minimă.

- Se consideră  $n$  ( $n \leq 100$ ) intervale reprezentând perioadele de viață a  $n$  scriitori. Să se realizeze un program care determină anul când se aflau în viață cei mai mulți scriitori.

- Într-un acvariu se găsesc  $n$  ( $n \leq 100$ ) pești carnivori, numerotăți de la 1 la  $n$ . Știind, pentru fiecare pește, care sunt peștii pe care îi poate mâncă, realizăți un program care determină ordinea în care aceștia se vor mâncă între ei, astfel încât la final, în acvariu, să rămână un număr minim de pești.

- Se consideră un graf neorientat cu  $n$  noduri. Să se determine o submulțime cu număr maxim de noduri, având proprietatea că oricare două noduri  $nu$  sunt adiacente.

- Se consideră un graf neorientat cu  $n$  noduri. Să se determine o submulțime cu număr maxim de noduri, având proprietatea că oricare două noduri sunt adiacente.

11. Se consideră un sir de  $N$  ( $N \leq 10000$ ) intervale de forma  $[A_i, B_i]$ , cu  $A_i, B_i$  numere întregi. Un din sirul celor  $N$  dacă există un alt interval care îl include strict pe acesta. Determinați numărul maxim de intervale care pot fi eliminate.

*Exemplu:* Pentru  $n=5$  și intervalele  $[0, 10], [2, 9], [3, 8], [1, 15], [6, 11]$ . Se va afișa: 3

12. Considerăm o tablă de săh de dimensiune  $n \times n$  ( $n \leq 10$ ). Să se determine un drum de lungime minimă prin care un cal aflat în poziția  $(x_i, y_i)$  se deplasează în  $(x_j, y_j)$ .

*Exemplu:* Pentru  $n=5$ , poziția de început  $(1,2)$  și poziția finală  $(2,5)$ . Se va afișa:  $(1,2)(3,3)(2,5)$

13. Se consideră  $n$  orașe ( $n \leq 100$ ). Se cunosc distanțele dintre oricare două orașe. Un comis-voiajor trebuie să treacă prin toate cele  $n$  orașe. Se cere să se determine un drum care pornește dintr-un oraș, trece exact o dată prin fiecare dintre celelalte orașe și apoi revine la primul oraș, astfel încât lungimea drumului să fie minimă.

14. Se consideră  $m$  ( $m \leq 10$ ) vectori  $V_1, V_2, \dots, V_m$ , care conțin  $n_1, n_2, \dots, n_m$  elemente, ordonate crescător. Se interclasă vectorii dați, obținându-se un vector de lungime  $n_1+n_2+\dots+n_m$  elemente, ordonate crescător. Se știe că interclasarea a doi vectori care conțin  $n_1$ , respectiv  $n_2$  elemente necesită un timp egal cu  $n_1+n_2$ . Să se determine ordinea optimă în care trebuie efectuată interclasarea tuturor vectorilor.

### 3.5. Probleme de concurs

#### 3.5.1 Probleme propuse

1. (\*\*\*\*) Pentru un număr  $n \leq 100$  dat, să se determine cel mai mic număr natural care are exact  $n$  divizori. Fișierul de intrare *divizori.in* conține, pe prima linie, numărul  $n$  al divizorilor pe care trebuie să îi aibă numărul căutat.

Fișierul de ieșire *divizori.out* trebuie să conțină o singură linie pe care se va afla cel mai mic număr natural care are exact  $n$  divizori.

*Exemplu:*

*divizori.in*

6

*divizori.out*

12

*Soluție:*

Dacă  $n$  este un număr natural și  $a$  este un sir care conține puterile la care apar factorii primi din descompunerea lui  $n$ , atunci numărul divizorilor lui  $n$  este:

$$Nr_{div} = \sum_{i=0}^k a_i + 1.$$

Demonstrația acestei teoreme este foarte simplă. Numărului  $n$  i se asociază sirul  $a_i$  al puterilor la care apar factorii primi din descompunerea sa. Astfel, fiecărui divizor

îi va corespunde un element al produsului cartezian  $X_1 * X_2 * \dots * X_k$ , unde  $X_i = \{0, 1, \dots, a_i\}$ , iar numărul elementelor acestui produs cartezian este exact cu  $Nr_{div}$ .

*Descompunerea problemei în subprobleme și structuri de date utilizate:*

De fapt, nu trebuie să fie luate în considerare toate numerele prime, fiind suficiente doar cele mai mici. În continuare se va presupune că s-au ales primele 15 numere prime. La început se determină vectorul  $d$  al divizorilor numărului  $n$ ; se consideră că acest vector are  $m$  elemente. Se construiește o matrice  $A$  cu 15 linii și  $m$  coloane,  $A_{ij}$  având valoarea egală cu cel mai mic număr care are ca factori primi primele  $i$  numere prime și având  $d_j$  divizori. Rezultatul cerut va fi valoarea minimă de pe ultima coloană a matricei.

*Relațiile de recurență:*

Primul element al fiecărei linii ( $A_{i,1}$ ) va avea valoarea 1, deoarece 1 este cel mai mic număr cu un singur divizor;  $A_{0,1}=\infty$  deoarece nu există numere cu 0 divizori și nici un factor prim. Elementul  $A_{i,j}$  poate fi calculat folosind valorile  $A_{i-1,k}$  pentru  $k < j$  și  $d_k$  este divizor al lui  $d_j$ .  $A_{i-1,k}$  are  $d_k$  divizori, deci o valoare  $A_{i,j}$  cu  $d_j$  divizori poate fi obținută calculând valoarea:

$$A_{i-1,k} * p_i^{d_j/d_k+1},$$

unde  $p_i$  este al  $i$ -lea număr prim. Valoarea  $A_{i,j}$  va fi atesa că fiind minimul acestor valori. Dacă această valoare minimă este mai mare decât  $A_{i-1,j}$ , atunci  $A_{i,j}$  va primi valoarea  $A_{i-1,j}$ . Deoarece ar trebui să se lucreze cu numere foarte mari, în program se va determina logaritmii naturali ai valorilor  $A_{i,j}$ , iar în final calculăm valoarea reală a minimului de pe ultima coloană a matricei, folosind operații cu numere mari.

```

1 const p:array[0..15] of integer;
2   (1,2,3,5,7,11,13,17,19,23,29,3
3   1,37,41,43,47);
4
5 type nr=array[0..100] of integer;
6 var n,i,j,k,nd:integer; rez:nr;
7 d:array[0..100] of integer;
8 a:array[0..15,0..100] of double;
9 t:array[0..15,0..100] of byte;
10 tmp:double;
11
12 procedure citire;
13 begin
14   assign(input,'divizori.in');
15   reset(input);
16   assign(output,'divizori.out');
17   rewrite(output);
18   readln(n);
19 end;
20
21 procedure inmul(var a:nr;b:integer);
22 var i,t:integer;
23 begin
24   t:=0; i:=1;
25   while i<=b do
26     begin
27       t:=t+a;
28       i:=i+1;
29     end;
30   a:=t;
31 end;
32
33 begin
34   read(nr);
35   for i:=1 to 15 do
36     begin
37       for j:=1 to nr do
38         begin
39           if j=1 then rez:=1
40           else rez:=rez*p[i]^(j-1);
41           if rez>nr then rez:=nr;
42           if rez<0 then rez:=0;
43           if rez<=nr then rez:=rez;
44           if rez>nr then rez:=nr;
45           if rez<0 then rez:=0;
46           if rez<=nr then rez:=rez;
47           if rez>nr then rez:=nr;
48           if rez<0 then rez:=0;
49           if rez<=nr then rez:=rez;
50           if rez>nr then rez:=nr;
51           if rez<0 then rez:=0;
52           if rez<=nr then rez:=rez;
53           if rez>nr then rez:=nr;
54           if rez<0 then rez:=0;
55           if rez<=nr then rez:=rez;
56           if rez>nr then rez:=nr;
57           if rez<0 then rez:=0;
58           if rez<=nr then rez:=rez;
59           if rez>nr then rez:=nr;
60           if rez<0 then rez:=0;
61           if rez<=nr then rez:=rez;
62           if rez>nr then rez:=nr;
63           if rez<0 then rez:=0;
64           if rez<=nr then rez:=rez;
65           if rez>nr then rez:=nr;
66           if rez<0 then rez:=0;
67           if rez<=nr then rez:=rez;
68           if rez>nr then rez:=nr;
69           if rez<0 then rez:=0;
70           if rez<=nr then rez:=rez;
71           if rez>nr then rez:=nr;
72           if rez<0 then rez:=0;
73           if rez<=nr then rez:=rez;
74           if rez>nr then rez:=nr;
75           if rez<0 then rez:=0;
76           if rez<=nr then rez:=rez;
77           if rez>nr then rez:=nr;
78           if rez<0 then rez:=0;
79           if rez<=nr then rez:=rez;
80           if rez>nr then rez:=nr;
81           if rez<0 then rez:=0;
82           if rez<=nr then rez:=rez;
83           if rez>nr then rez:=nr;
84           if rez<0 then rez:=0;
85           if rez<=nr then rez:=rez;
86           if rez>nr then rez:=nr;
87           if rez<0 then rez:=0;
88           if rez<=nr then rez:=rez;
89           if rez>nr then rez:=nr;
90           if rez<0 then rez:=0;
91           if rez<=nr then rez:=rez;
92           if rez>nr then rez:=nr;
93           if rez<0 then rez:=0;
94           if rez<=nr then rez:=rez;
95           if rez>nr then rez:=nr;
96           if rez<0 then rez:=0;
97           if rez<=nr then rez:=rez;
98           if rez>nr then rez:=nr;
99           if rez<0 then rez:=0;
100          if rez<=nr then rez:=rez;
101          if rez>nr then rez:=nr;
102          if rez<0 then rez:=0;
103          if rez<=nr then rez:=rez;
104          if rez>nr then rez:=nr;
105          if rez<0 then rez:=0;
106          if rez<=nr then rez:=rez;
107          if rez>nr then rez:=nr;
108          if rez<0 then rez:=0;
109          if rez<=nr then rez:=rez;
110          if rez>nr then rez:=nr;
111          if rez<0 then rez:=0;
112          if rez<=nr then rez:=rez;
113          if rez>nr then rez:=nr;
114          if rez<0 then rez:=0;
115          if rez<=nr then rez:=rez;
116          if rez>nr then rez:=nr;
117          if rez<0 then rez:=0;
118          if rez<=nr then rez:=rez;
119          if rez>nr then rez:=nr;
120          if rez<0 then rez:=0;
121          if rez<=nr then rez:=rez;
122          if rez>nr then rez:=nr;
123          if rez<0 then rez:=0;
124          if rez<=nr then rez:=rez;
125          if rez>nr then rez:=nr;
126          if rez<0 then rez:=0;
127          if rez<=nr then rez:=rez;
128          if rez>nr then rez:=nr;
129          if rez<0 then rez:=0;
130          if rez<=nr then rez:=rez;
131          if rez>nr then rez:=nr;
132          if rez<0 then rez:=0;
133          if rez<=nr then rez:=rez;
134          if rez>nr then rez:=nr;
135          if rez<0 then rez:=0;
136          if rez<=nr then rez:=rez;
137          if rez>nr then rez:=nr;
138          if rez<0 then rez:=0;
139          if rez<=nr then rez:=rez;
140          if rez>nr then rez:=nr;
141          if rez<0 then rez:=0;
142          if rez<=nr then rez:=rez;
143          if rez>nr then rez:=nr;
144          if rez<0 then rez:=0;
145          if rez<=nr then rez:=rez;
146          if rez>nr then rez:=nr;
147          if rez<0 then rez:=0;
148          if rez<=nr then rez:=rez;
149          if rez>nr then rez:=nr;
150          if rez<0 then rez:=0;
151          if rez<=nr then rez:=rez;
152          if rez>nr then rez:=nr;
153          if rez<0 then rez:=0;
154          if rez<=nr then rez:=rez;
155          if rez>nr then rez:=nr;
156          if rez<0 then rez:=0;
157          if rez<=nr then rez:=rez;
158          if rez>nr then rez:=nr;
159          if rez<0 then rez:=0;
160          if rez<=nr then rez:=rez;
161          if rez>nr then rez:=nr;
162          if rez<0 then rez:=0;
163          if rez<=nr then rez:=rez;
164          if rez>nr then rez:=nr;
165          if rez<0 then rez:=0;
166          if rez<=nr then rez:=rez;
167          if rez>nr then rez:=nr;
168          if rez<0 then rez:=0;
169          if rez<=nr then rez:=rez;
170          if rez>nr then rez:=nr;
171          if rez<0 then rez:=0;
172          if rez<=nr then rez:=rez;
173          if rez>nr then rez:=nr;
174          if rez<0 then rez:=0;
175          if rez<=nr then rez:=rez;
176          if rez>nr then rez:=nr;
177          if rez<0 then rez:=0;
178          if rez<=nr then rez:=rez;
179          if rez>nr then rez:=nr;
180          if rez<0 then rez:=0;
181          if rez<=nr then rez:=rez;
182          if rez>nr then rez:=nr;
183          if rez<0 then rez:=0;
184          if rez<=nr then rez:=rez;
185          if rez>nr then rez:=nr;
186          if rez<0 then rez:=0;
187          if rez<=nr then rez:=rez;
188          if rez>nr then rez:=nr;
189          if rez<0 then rez:=0;
190          if rez<=nr then rez:=rez;
191          if rez>nr then rez:=nr;
192          if rez<0 then rez:=0;
193          if rez<=nr then rez:=rez;
194          if rez>nr then rez:=nr;
195          if rez<0 then rez:=0;
196          if rez<=nr then rez:=rez;
197          if rez>nr then rez:=nr;
198          if rez<0 then rez:=0;
199          if rez<=nr then rez:=rez;
200          if rez>nr then rez:=nr;
201          if rez<0 then rez:=0;
202          if rez<=nr then rez:=rez;
203          if rez>nr then rez:=nr;
204          if rez<0 then rez:=0;
205          if rez<=nr then rez:=rez;
206          if rez>nr then rez:=nr;
207          if rez<0 then rez:=0;
208          if rez<=nr then rez:=rez;
209          if rez>nr then rez:=nr;
210          if rez<0 then rez:=0;
211          if rez<=nr then rez:=rez;
212          if rez>nr then rez:=nr;
213          if rez<0 then rez:=0;
214          if rez<=nr then rez:=rez;
215          if rez>nr then rez:=nr;
216          if rez<0 then rez:=0;
217          if rez<=nr then rez:=rez;
218          if rez>nr then rez:=nr;
219          if rez<0 then rez:=0;
220          if rez<=nr then rez:=rez;
221          if rez>nr then rez:=nr;
222          if rez<0 then rez:=0;
223          if rez<=nr then rez:=rez;
224          if rez>nr then rez:=nr;
225          if rez<0 then rez:=0;
226          if rez<=nr then rez:=rez;
227          if rez>nr then rez:=nr;
228          if rez<0 then rez:=0;
229          if rez<=nr then rez:=rez;
230          if rez>nr then rez:=nr;
231          if rez<0 then rez:=0;
232          if rez<=nr then rez:=rez;
233          if rez>nr then rez:=nr;
234          if rez<0 then rez:=0;
235          if rez<=nr then rez:=rez;
236          if rez>nr then rez:=nr;
237          if rez<0 then rez:=0;
238          if rez<=nr then rez:=rez;
239          if rez>nr then rez:=nr;
240          if rez<0 then rez:=0;
241          if rez<=nr then rez:=rez;
242          if rez>nr then rez:=nr;
243          if rez<0 then rez:=0;
244          if rez<=nr then rez:=rez;
245          if rez>nr then rez:=nr;
246          if rez<0 then rez:=0;
247          if rez<=nr then rez:=rez;
248          if rez>nr then rez:=nr;
249          if rez<0 then rez:=0;
250          if rez<=nr then rez:=rez;
251          if rez>nr then rez:=nr;
252          if rez<0 then rez:=0;
253          if rez<=nr then rez:=rez;
254          if rez>nr then rez:=nr;
255          if rez<0 then rez:=0;
256          if rez<=nr then rez:=rez;
257          if rez>nr then rez:=nr;
258          if rez<0 then rez:=0;
259          if rez<=nr then rez:=rez;
260          if rez>nr then rez:=nr;
261          if rez<0 then rez:=0;
262          if rez<=nr then rez:=rez;
263          if rez>nr then rez:=nr;
264          if rez<0 then rez:=0;
265          if rez<=nr then rez:=rez;
266          if rez>nr then rez:=nr;
267          if rez<0 then rez:=0;
268          if rez<=nr then rez:=rez;
269          if rez>nr then rez:=nr;
270          if rez<0 then rez:=0;
271          if rez<=nr then rez:=rez;
272          if rez>nr then rez:=nr;
273          if rez<0 then rez:=0;
274          if rez<=nr then rez:=rez;
275          if rez>nr then rez:=nr;
276          if rez<0 then rez:=0;
277          if rez<=nr then rez:=rez;
278          if rez>nr then rez:=nr;
279          if rez<0 then rez:=0;
280          if rez<=nr then rez:=rez;
281          if rez>nr then rez:=nr;
282          if rez<0 then rez:=0;
283          if rez<=nr then rez:=rez;
284          if rez>nr then rez:=nr;
285          if rez<0 then rez:=0;
286          if rez<=nr then rez:=rez;
287          if rez>nr then rez:=nr;
288          if rez<0 then rez:=0;
289          if rez<=nr then rez:=rez;
290          if rez>nr then rez:=nr;
291          if rez<0 then rez:=0;
292          if rez<=nr then rez:=rez;
293          if rez>nr then rez:=nr;
294          if rez<0 then rez:=0;
295          if rez<=nr then rez:=rez;
296          if rez>nr then rez:=nr;
297          if rez<0 then rez:=0;
298          if rez<=nr then rez:=rez;
299          if rez>nr then rez:=nr;
300          if rez<0 then rez:=0;
301          if rez<=nr then rez:=rez;
302          if rez>nr then rez:=nr;
303          if rez<0 then rez:=0;
304          if rez<=nr then rez:=rez;
305          if rez>nr then rez:=nr;
306          if rez<0 then rez:=0;
307          if rez<=nr then rez:=rez;
308          if rez>nr then rez:=nr;
309          if rez<0 then rez:=0;
310          if rez<=nr then rez:=rez;
311          if rez>nr then rez:=nr;
312          if rez<0 then rez:=0;
313          if rez<=nr then rez:=rez;
314          if rez>nr then rez:=nr;
315          if rez<0 then rez:=0;
316          if rez<=nr then rez:=rez;
317          if rez>nr then rez:=nr;
318          if rez<0 then rez:=0;
319          if rez<=nr then rez:=rez;
320          if rez>nr then rez:=nr;
321          if rez<0 then rez:=0;
322          if rez<=nr then rez:=rez;
323          if rez>nr then rez:=nr;
324          if rez<0 then rez:=0;
325          if rez<=nr then rez:=rez;
326          if rez>nr then rez:=nr;
327          if rez<0 then rez:=0;
328          if rez<=nr then rez:=rez;
329          if rez>nr then rez:=nr;
330          if rez<0 then rez:=0;
331          if rez<=nr then rez:=rez;
332          if rez>nr then rez:=nr;
333          if rez<0 then rez:=0;
334          if rez<=nr then rez:=rez;
335          if rez>nr then rez:=nr;
336          if rez<0 then rez:=0;
337          if rez<=nr then rez:=rez;
338          if rez>nr then rez:=nr;
339          if rez<0 then rez:=0;
340          if rez<=nr then rez:=rez;
341          if rez>nr then rez:=nr;
342          if rez<0 then rez:=0;
343          if rez<=nr then rez:=rez;
344          if rez>nr then rez:=nr;
345          if rez<0 then rez:=0;
346          if rez<=nr then rez:=rez;
347          if rez>nr then rez:=nr;
348          if rez<0 then rez:=0;
349          if rez<=nr then rez:=rez;
350          if rez>nr then rez:=nr;
351          if rez<0 then rez:=0;
352          if rez<=nr then rez:=rez;
353          if rez>nr then rez:=nr;
354          if rez<0 then rez:=0;
355          if rez<=nr then rez:=rez;
356          if rez>nr then rez:=nr;
357          if rez<0 then rez:=0;
358          if rez<=nr then rez:=rez;
359          if rez>nr then rez:=nr;
360          if rez<0 then rez:=0;
361          if rez<=nr then rez:=rez;
362          if rez>nr then rez:=nr;
363          if rez<0 then rez:=0;
364          if rez<=nr then rez:=rez;
365          if rez>nr then rez:=nr;
366          if rez<0 then rez:=0;
367          if rez<=nr then rez:=rez;
368          if rez>nr then rez:=nr;
369          if rez<0 then rez:=0;
370          if rez<=nr then rez:=rez;
371          if rez>nr then rez:=nr;
372          if rez<0 then rez:=0;
373          if rez<=nr then rez:=rez;
374          if rez>nr then rez:=nr;
375          if rez<0 then rez:=0;
376          if rez<=nr then rez:=rez;
377          if rez>nr then rez:=nr;
378          if rez<0 then rez:=0;
379          if rez<=nr then rez:=rez;
380          if rez>nr then rez:=nr;
381          if rez<0 then rez:=0;
382          if rez<=nr then rez:=rez;
383          if rez>nr then rez:=nr;
384          if rez<0 then rez:=0;
385          if rez<=nr then rez:=rez;
386          if rez>nr then rez:=nr;
387          if rez<0 then rez:=0;
388          if rez<=nr then rez:=rez;
389          if rez>nr then rez:=nr;
390          if rez<0 then rez:=0;
391          if rez<=nr then rez:=rez;
392          if rez>nr then rez:=nr;
393          if rez<0 then rez:=0;
394          if rez<=nr then rez:=rez;
395          if rez>nr then rez:=nr;
396          if rez<0 then rez:=0;
397          if rez<=nr then rez:=rez;
398          if rez>nr then rez:=nr;
399          if rez<0 then rez:=0;
400          if rez<=nr then rez:=rez;
401          if rez>nr then rez:=nr;
402          if rez<0 then rez:=0;
403          if rez<=nr then rez:=rez;
404          if rez>nr then rez:=nr;
405          if rez<0 then rez:=0;
406          if rez<=nr then rez:=rez;
407          if rez>nr then rez:=nr;
408          if rez<0 then rez:=0;
409          if rez<=nr then rez:=rez;
410          if rez>nr then rez:=nr;
411          if rez<0 then rez:=0;
412          if rez<=nr then rez:=rez;
413          if rez>nr then rez:=nr;
414          if rez<0 then rez:=0;
415          if rez<=nr then rez:=rez;
416          if rez>nr then rez:=nr;
417          if rez<0 then rez:=0;
418          if rez<=nr then rez:=rez;
419          if rez>nr then rez:=nr;
420          if rez<0 then rez:=0;
421          if rez<=nr then rez:=rez;
422          if rez>nr then rez:=nr;
423          if rez<0 then rez:=0;
424          if rez<=nr then rez:=rez;
425          if rez>nr then rez:=nr;
426          if rez<0 then rez:=0;
427          if rez<=nr then rez:=rez;
428          if rez>nr then rez:=nr;
429          if rez<0 then rez:=0;
430          if rez<=nr then rez:=rez;
431          if rez>nr then rez:=nr;
432          if rez<0 then rez:=0;
433          if rez<=nr then rez:=rez;
434          if rez>nr then rez:=nr;
435          if rez<0 then rez:=0;
436          if rez<=nr then rez:=rez;
437          if rez>nr then rez:=nr;
438          if rez<0 then rez:=0;
439          if rez<=nr then rez:=rez;
440          if rez>nr then rez:=nr;
441          if rez<0 then rez:=0;
442          if rez<=nr then rez:=rez;
443          if rez>nr then rez:=nr;
444          if rez<0 then rez:=0;
445          if rez<=nr then rez:=rez;
446          if rez>nr then rez:=nr;
447          if rez<0 then rez:=0;
448          if rez<=nr then rez:=rez;
449          if rez>nr then rez:=nr;
450          if rez<0 then rez:=0;
451          if rez<=nr then rez:=rez;
452          if rez>nr then rez:=nr;
453          if rez<0 then rez:=0;
454          if rez<=nr then rez:=rez;
455          if rez>nr then rez:=nr;
456          if rez<0 then rez:=0;
457          if rez<=nr then rez:=rez;
458          if rez>nr then rez:=nr;
459          if rez<0 then rez:=0;
460          if rez<=nr then rez:=rez;
461          if rez>nr then rez:=nr;
462          if rez<0 then rez:=0;
463          if rez<=nr then rez:=rez;
464          if rez>nr then rez:=nr;
465          if rez<0 then rez:=0;
466          if rez<=nr then rez:=rez;
467          if rez>nr then rez:=nr;
468          if rez<0 then rez:=0;
469          if rez<=nr then rez:=rez;
470          if rez>nr then rez:=nr;
471          if rez<0 then rez:=0;
472          if rez<=nr then rez:=rez;
473          if rez>nr then rez:=nr;
474          if rez<0 then rez:=0;
475          if rez<=nr then rez:=rez;
476          if rez>nr then rez:=nr;
477          if rez<0 then rez:=0;
478          if rez<=nr then rez:=rez;
479          if rez>nr then rez:=nr;
480          if rez<0 then rez:=0;
481          if rez<=nr then rez:=rez;
482          if rez>nr then rez:=nr;
483          if rez<0 then rez:=0;
484          if rez<=nr then rez:=rez;
485          if rez>nr then rez:=nr;
486          if rez<0 then rez:=0;
487          if rez<=nr then rez:=rez;
488          if rez>nr then rez:=nr;
489          if rez<0 then rez:=0;
490          if rez<=nr then rez:=rez;
491          if rez>nr then rez:=nr;
492          if rez<0 then rez:=0;
493          if rez<=nr then rez:=rez;
494          if rez>nr then rez:=nr;
495          if rez<0 then rez:=0;
496          if rez<=nr then rez:=rez;
497          if rez>nr then rez:=nr;
498          if rez<0 then rez:=0;
499          if rez<=nr then rez:=rez;
500          if rez>nr then rez:=nr;
501          if rez<0 then rez:=0;
502          if rez<=nr then rez:=rez;
503          if rez>nr then rez:=nr;
504          if rez<0 then rez:=0;
505          if rez<=nr then rez:=rez;
506          if rez>nr then rez:=nr;
507          if rez<0 then rez:=0;
508          if rez<=nr then rez:=rez;
509          if rez>nr then rez:=nr;
510          if rez<0 then rez:=0;
511          if rez<=nr then rez:=rez;
512          if rez>nr then rez:=nr;
513          if rez<0 then rez:=0;
514          if rez<=nr then rez:=rez;
515          if rez>nr then rez:=nr;
516          if rez<0 then rez:=0;
517          if rez<=nr then rez:=rez;
518          if rez>nr then rez:=nr;
519          if rez<0 then rez:=0;
520          if rez<=nr then rez:=rez;
521          if rez>nr then rez:=nr;
522          if rez<0 then rez:=0;
523          if rez<=nr then rez:=rez;
524          if rez>nr then rez:=nr;
525          if rez<0 then rez:=0;
526          if rez<=nr then rez:=rez;
527          if rez>nr then rez:=nr;
528          if rez<0 then rez:=0;
529          if rez<=nr then rez:=rez;
530          if rez>nr then rez:=nr;
531          if rez<0 then rez:=0;
532          if rez<=nr then rez:=rez;
533          if rez>nr then rez:=nr;
534          if rez<0 then rez:=0;
535          if rez<=nr then rez:=rez;
536          if rez>nr then rez:=nr;
537          if rez<0 then rez:=0;
538          if rez<=nr then rez:=rez;
539          if rez>nr then rez:=nr;
540          if rez<0 then rez:=0;
541          if rez<=nr then rez:=rez;
542          if rez>nr then rez:=nr;
543          if rez<0 then rez:=0;
544          if rez<=nr then rez:=rez;
545          if rez>nr then rez:=nr;
546          if rez<0 then rez:=0;
547          if rez<=nr then rez:=rez;
548          if rez>nr then rez:=nr;
549          if rez<0 then rez:=0;
550          if rez<=nr then rez:=rez;
551          if rez>nr then rez:=nr;
552          if rez<0 then rez:=0;
553          if rez<=nr then rez:=rez;
554          if rez>nr then rez:=nr;
555          if rez<0 then rez:=0;
556          if rez<=nr then rez:=rez;
557          if rez>nr then rez:=nr;
558          if rez<0 then rez:=0;
559          if rez<=nr then rez:=rez;
560          if rez>nr then rez:=nr;
561          if rez<0 then rez:=0;
562          if rez<=nr then rez:=rez;
563          if rez>nr then rez:=nr;
564          if rez<0 then rez:=0;
565          if rez<=nr then rez:=rez;
566          if rez>nr then rez:=nr;
567          if rez<0 then rez:=0;
568          if rez<=nr then rez:=rez;
569          if rez>nr then rez:=nr;
570          if rez<0 then rez:=0;
571          if rez<=nr then rez:=rez;
572          if rez>nr then rez:=nr;
573          if rez<0 then rez:=0;
574          if rez<=nr then rez:=rez;
575          if rez>nr then rez:=nr;
576          if rez<0 then rez:=0;
577          if rez<=nr then rez:=rez;
578          if rez>nr then rez:=nr;
579          if rez<0 then rez:=0;
580          if rez<=nr then rez:=rez;
581          if rez>nr then rez:=nr;
582          if rez<0 then rez:=0;
583          if rez<=nr then rez:=rez;
584          if rez>nr then rez:=nr;
585          if rez<0 then rez:=0;
586          if rez<=nr then rez:=rez;
587          if rez>nr then rez:=nr;
588          if rez<0 then rez:=0;
589          if rez<=nr then rez:=rez;
590          if rez>nr then rez:=nr;
591          if rez<0 then rez:=0;
592          if rez<=nr then rez:=rez;
593          if rez>nr then rez:=nr;
594          if rez<0 then rez:=0;
595          if rez<=nr then rez:=rez;
596          if rez>nr then rez:=nr;
597          if rez<0 then rez:=0;
598          if rez<=nr then rez:=rez;
599          if rez>nr then rez:=nr;
600          if rez<0 then rez:=0;
601          if rez<=nr then rez:=rez;
602          if rez>nr then rez:=nr;
603          if rez<0 then rez:=0;
604          if rez<=nr then rez:=rez;
605          if rez>nr then rez:=nr;
606          if rez<0 then rez:=0;
607          if rez<=nr then rez:=rez;
608          if rez>nr then rez:=nr;
609          if rez<0 then rez:=0;
610          if rez<=nr then rez:=rez;
611          if rez>nr then rez:=nr;
612          if rez<0 then rez:=0;
613          if rez<=nr then rez:=rez;
6
```

```

25 while (i<=a[0]) or (t>0) do begin
26   inc(t,a[i]*b);
27   a[i]:=t mod 10;
28   t:=t div 10;
29   inc(i);
30 end;
31 a[10]:=i-1;
32 end;
33
34 procedure refa(i,j:integer);
35 var k,it:integer;
36 begin
37   k:=t[i,j];
38   if i=0 then exit;
39   for it:=1 to (d[j]-1) do
40     inmul(rez,p[it]);
41   refa(i-1,k);
42 end;
43
44 begin
45   citire;
46   for i:=1 to n do
47     if n mod i=0 then begin
48       d[nd]:=i;
49       inc(nd);
50     end;
51   a[0,0]:=0.0;
52   for i:=1 to nd-1 do a[0,i]:=1e13;
53   for i:=1 to 15 do begin
54     a[i,0]:=0.0;
55     t[i,0]:=0;
56     for j:=1 to nd-1 do begin
57       a[i,j]:=a[i-1,j];
58       t[i,j]:=j;
59       for k:=0 to j-1 do begin
60         if d[j] mod d[k]<>0 then
61           continue;
62         tmp:=a[i-1,k]+
63             (d[j] div d[k]-1)*ln(p[i]);
64         if a[i,j]>tmp then
65           begin
66             a[i,j]:=tmp;
67             t[i,j]:=k;
68           end;
69         end;
70       end;
71     rez[0]:=1;
72     rez[1]:=1;
73     refa(15,nd-1);
74     for i:=rez[0] downto 1 do
75       write(rez[i]);
76     writeln;
77   end;
78 end.

```

**2. (\*\*\*)** *Bronzărel* se antrenează zi de zi pentru a deveni un mare olimpic de informatică, avându-l pe *Zăhărel* ca mentor/guru. Desigur, sursa lor preferată de probleme este info-arena! Astăzi, *Bronzărel* a rezolvat seria de probleme "Secvența X" ( $X = 1, 2, 3, \dots$ ). Văzându-l foarte încrezător, *Zăhărel* vrea să-l testeze pe *Bronzărel* cu o nouă problemă care nu este pe info-arena și îi spune: "Îți dau un sir de  $N \leq 5.000$  numere întregi și vreau să-mi spui care este cel mai scurt subșir crescător maximal. La probleme cu secvențe te-ai descurcat, vei reuși și la subșiruri?"

Spunem că un subșir  $B=(a_1, a_2, \dots, a_k)$  este crescător maximal dacă  $a_1 \leq a_2 \leq \dots \leq a_k$  și nu există nici un  $x$  astfel încât: să existe  $j < K$ ,  $i_j < x < i_{j+1}$  și  $a_{i_j} \leq a_x \leq a_{i_{j+1}}$ , sau  $1 \leq x < i_1$  și  $a_x \leq a_1$ , sau  $i_K < x \leq N$  și  $a_{i_K} \leq a_x$ . Imagineați-vă că sunteți în locul lui *Bronzărel* și scrieți un program care rezolvă problema primită.

Pe prima linie din fișierul *subsir2.in* se va găsi numărul  $N$ . Pe următoarea linie vor fi scrise  $N$  numere întregi. Sirul dat va conține numere întregi din intervalul  $[-1.000.000, 1.000.000]$ .

Prima linie din fișierul *subsir2.out* va conține un număr  $L_{\min}$ , reprezentând lungimea minimă a unui subșir crescător maximal. Pe următoarea linie se vor afișa  $L_{\min}$  numere în ordine crescătoare, reprezentând pozițiile elementelor din sirul inițial care fac parte din subșirul ales. Dacă există mai multe soluții, se va afișa cea lexicografic minimă, din punct de vedere al valorilor elementelor din subșir.

*Exemplu:*

<i>subsir2.in</i> 6 1 3 6 2 5 4
---------------------------------------

<i>subsir2.out</i> 3 1 4 6
----------------------------------

*Soluție:*

Problema poate fi considerată asemănătoare cu cea a celui mai lung subșir crescător, având o rezolvare similară de complexitate  $O(N^2)$ , folosind metoda programării dinamice.

*Descompunerea problemei în subprobleme și structuri de date utilizate:*

Se vor construi doi vectori:

$BST[i]$  = lungimea celui mai scurt subșir crescător maximal care începe cu poziția  $i$   
 $T[i]$  = următorul element după poziția  $i$  în cel mai scurt subșir crescător maximal care începe cu  $i$  (pentru reconstituire).

*Relațiile de recurență:*

Pentru fiecare  $i$ , se va căuta un  $j > i$  astfel încât  $V[i] \leq V[j]$  (unde  $V$  este vectorul de numere) și se alege acela cu  $BST[j]$  minim,  $BST[i]$  devenind  $BST[j]+1$ , iar  $T[i]$  devine  $j$ . Dacă nu există nici un  $j$ ,  $BST[i]$  se initializează cu 1 și  $T[i]$  cu  $i$ .

Ca sirul construit să fie maximal trebuie ca atunci când verificăm  $j$ -urile pentru un  $i$  fixat,  $j$ -ul respectiv să fie "dominant", în sensul că să nu existe un  $i < k < j$  astfel încat  $V[i] \leq V[k] \leq V[j]$ , deoarece sirul construit cu primul element în  $i$  și al doilea în  $j$  ar putea fi extins inserând  $k$  între  $i$  și  $j$ . Pentru a face verificarea în  $O(1)$ , facem observația că ne interesează doar acel  $V[k]$  minim care este  $\geq V[i]$ . Pe măsură ce se avansează cu variabila  $j$ , se păstrează o variabilă  $min$ , reprezentând minimul dintre valorile  $V[j]$  parcuse până acum, care sunt  $\geq V[i]$ .

Astfel, când se ajunge la un  $j$ , se verifică înainte dacă  $V[j] < min$  (condiție necesară pentru a construi un sir maximal). Un alt detaliu în soluție este obținerea soluției minime din punct de vedere lexicografic. Când se selectează  $j$ -ul pentru fiecare  $i$ , pe lângă faptul că se alege acela cu  $BST[ij]$  minim, în caz de egalitate se va alege acela cu valoarea  $V[j]$  minimă. Selectarea este corectă deoarece nu vor exista  $j_1, j_2$  cu  $BST[ij_1] = BST[ij_2]$  minim, iar  $V[j_1] = V[j_2]$ .

```

1 const inf=16191;
2
3 var n,rez,poz,min,i,j:integer;
4 a,bst,t:array[0..5000] of integer;
5 ok:array[0..5000] of boolean;
6
7 procedure citire;
8 var i:integer;
9 begin
10 assign(input,'subsir2.in');
11 reset(input);
12 assign(output,'subsir2.out');
13 rewrite(output);
14 readln(n); min:=inf;
15 for i:=0 to n-1 do begin
16   read(a[i]);
17   if min<=a[i] then continue;
18   ok[i]:=true; min:=a[i];
19 end;
20 end;
21
22 begin
23   citire; rez:=inf;
24   for i:=n-1 downto 0 do begin
25     bst[i]:=inf; min:=inf; t[i]:=-1;
26     for j:=i+1 to n-1 do begin
27       if a[j]<a[i] then continue;
28       if (min>a[j])and((bst[i]>bst[j]+1)
29       or((bst[i]=bst[j]+1)and
30       (a[j]<=t[i]))) then begin
31         bst[i]:=bst[j]+1; t[i]:=j;
32       end;
33       if min>a[j] then min:=a[j];
34     end;
35     if bst[i]=inf then begin
36       bst[i]:=1; t[i]:=i;
37     end;
38     if ok[i] and((rez>bst[i])or
39     ((rez=bst[i])and(a[i]<=a[poz]))) then begin
40       rez:=bst[i]; poz:=i;
41     end;
42   end;
43   writeln(rez); i:=poz;
44   while i<>t[i] do begin
45     write(i+1,' '); i:=t[i];
46   end; writeln(i+1);
47 end.
48

```

```

#include <stdio.h>
#define INF 0x3f3f3f

int N, A[5001], bst[5001],
T[5001], rez=INF, poz,min;
char ok[5001];

void citire() {
    int i;
    freopen("subsir2.in","r",stdin);
    freopen("subsir2.out","w",stdout);
    scanf("%d", &N); min=INF;
    for (i=0; i < N; i++) {
        scanf("%d", &A[i]);
        if (min <= A[i]) continue;
        ok[i]=1; min=A[i];
    }
}

int main()
{
    int i, j, min;
    citire();
    for (i=N-1; i >= 0; i--) {
        bst[i]=min=INF; T[i]=-1;
        for (j=i+1; j<N; j++)
        {
            if (A[j] < A[i]) continue;
            if (min>A[j]&&(bst[i]>bst[j]+1|
(bst[i]==bst[j]+1)&&A[j]<A[T[i]]))
            { bst[i]=bst[j]+1; T[i]=j; }
            if (min > A[j]) min=A[j];
        }
        if (bst[i]==INF)
        { bst[i]=1; T[i]=i; }
        if (ok[i]&&(rez>bst[i])|
(rez==bst[i]&&A[i]<=a[poz])))
        {
            rez=bst[i]; poz=i;
        }
    }
    printf("%d\n", rez);
    for (i=poz; i!=T[i]; i=T[i])
        printf("%d ", i+1);
    printf("%d\n", i+1);
    return 0;
}

```

3. (\*\*\*) Se consideră două siruri de  $N$ , respectiv  $M$  caractere ( $N, M \leq 100$ ). Asupra primului cuvânt se pot face următoarele schimbări în aşa fel încât acesta să ajungă identic cu al doilea:

- se poate șterge o literă;
- se poate insera o literă;
- se poate schimba o literă în altă literă.

Să se determine numărul minim de schimbări necesare pentru a obține două siruri identice. Din fișierul *editare.in* se vor citi cele două siruri și rezultatul va fi afișat în *editare.out*.

*Exemplu:*

*editare.in*

ana  
alla

*editare.out*

2

*Soluție:*

*Descompunerea problemei în subprobleme:*

O subproblemă se referă la determinarea numărului minim de schimbări necesare pentru a "potrivii" primele  $i$  valori din primul sir și primele  $j$  valori din al doilea sir

*Structuri de date utilizate:*

- Tablourile  $A(N)$  și  $B(M)$  rețin cele două siruri;
- Tabloul  $C(N,M)$  va memora succesiv soluțiile subproblemelor. Astfel elementul  $C[i,j]=k$ , semnifică faptul că numărul minim de schimbări necesare pentru a potrivii primele  $i$  valori din primul sir și primele  $j$  valori din al doilea sir este  $k$ .

*Relațiile de recurență:*

- $C[0,0] = 0$ ;
- $C[0, j] = j$ ;  $C[i, 0] = i$ ;
- $C[i,j] = \min(C[i-1,j], C[i,j-1], C[i-1,j-1])+1$  dacă  $A[i] \neq B[j]$   
 $C[i-1,j-1]$  dacă  $A[i] = B[j]$

Algoritmul prezentat are complexitatea  $O(n*m)$ .

```

1 var n,m,i,j:integer;a,b:string;
2 c:array[0..100,0..100] of integer;
3
4 procedure citire;
5 begin
6   assign(input,'editare.in');
7   reset(input);
8   assign(output,'editare.out');
9   rewrite(output);
10  readln(a); readln(b);
11  n:=length(a);
12  m:=length(b);
13  end;

```

```

14 function min(a,b:integer):integer;
15 begin
16   if a<b then min:=a
17     else min:=b;
18 end;
19
20 procedure solve;
21 begin
22   for i:=1 to n do c[i,0]:=i;
23   for j:=1 to m do c[0,j]:=j;
24   for i:=1 to n do
25     for j:=1 to m do
26       if a[i]=b[j] then
27         c[i,j]:=c[i-1,j-1]
28       else
29         c[i,j]:=min(min(c[i-1,j],
30                       c[i,j-1]),c[i-1,j-1])+1;
31   writeln(c[n,m]);
32 end;
33
34 begin
35   citire();
36   solve();
37   close(output);
38 end.

```

```

int min(int a,int b) {
  if (a < b) return a;
  return b;
}

void solve() {
  for (i=1;i<=n;i++) c[i][0]=i;
  for (j=1;j<=m;j++) c[0][j]=j;
  for (i=1;i<=n;i++)
    for (j=1;j<=m;j++)
      if (a[i]==b[j])
        c[i][j]=c[i-1][j-1];
      else
        c[i][j]=min(min(c[i-1][j],
                           c[i][j-1]),c[i-1][j-1])+1;
  printf("%d\n", c[n][m]);
}

int main() {
  citire();
  solve();
  return 0;
}

```

4. (\*\*\*\*) Se consideră un sir  $V$  ce conține  $N \leq 10.000$  elemente întregi. Să se realizeze un program care determină un subșir crescător de lungime maximă al acestuia. Un subșir crescător al lui  $V$  se poate descrie astfel:  $V_{i_1} \leq V_{i_2} \leq \dots \leq V_{i_k}$  și  $1 \leq i_1 < i_2 < \dots < i_k \leq N$ .

Din fișierul *subsr.in* se vor citi  $N$  și sirul  $V$ , iar rezultatul va fi afișat în *subsr.out*.

*Exemplu:*

*subsr.in*

8  
2 1 3 2 4 8 6 7

*subsr.out*

5  
1 2 4 6 7

*Soluție:*

*Descompunerea problemei în subprobleme și structuri de date utilizate:*

Fie  $V$  vectorul citit. Soluția pornește de la observația că ultimul element al unui subșir de lungime  $i$  este cel puțin la fel de mare ca ultimul element al subșirului candidat de lungime  $i-1$ . Se parcurge vectorul de la stânga la dreapta și se construiesc în paralel doi vectori  $P$  și  $Q$ , astfel: inițial vectorul  $Q$  este vid. Se ia fiecare element din  $V$  și se suprascrie peste cel mai mic element din  $Q$  care este strict mai mare ca el. Dacă nu există un asemenea element în  $Q$ , cu alte cuvinte dacă elementul analizat din  $V$  este mai mare ca toate elementele din  $Q$ , atunci el este adăugat la sfârșitul vectorului  $Q$ . Concomitent, se notează în vectorul  $P$  poziția pe care a fost adăugat în vectorul  $Q$  elementul din  $V$ . Astfel, în  $Q[i]$  se menține pe parcurs cel mai mic ultim element pentru un subșir de lungime  $i$ .

Lungimea  $L$  la care ajunge vectorul  $Q$  la sfârșitul acestei prelucrări este tocmai lungimea celui mai lung subșir crescător al vectorului  $V$ . Pentru a afla exact care sunt elementele subșirului crescător se procedează astfel: se caută ultima apariție în vectorul  $P$  a valorii  $L$ . Să spunem că ea este găsită pe poziția  $K_L$ . Se caută apoi ultima apariție în vectorul  $P$  a valorii  $L-1$ , anterior poziției  $K_L$ . Ea va fi pe poziția  $K_{L-1} < K_L$ . Analog se caută în vectorul  $P$  valorile  $L-2, L-3, \dots, 2, 1$ . Subșirul crescător este  $S=(V[K_1], V[K_2], \dots, V[K_L])$ .

Algoritmul are complexitatea  $O(N^2 \log N)$  deoarece inserția elementelor în  $Q$  se poate face în  $O(\log N)$  folosind o căutare binară.

```

1 const inf=16191;
2 var n,l,i:integer;
3 v,p,q,s:array[1..10000]of integer;
4 procedure citire;
5 begin
6   assign(input,'subsr.in');
7   reset(input);
8   assign(output,'subsr.out');
9   rewrite(output); readln(n);
10  for i:=1 to n do read(v[i]);
11 end;
12
13 function
14 insert(k,st,dr:integer):integer;
15 var m:integer;
16 begin
17   m:=(st+dr)div 2;
18   if st=dr then begin
19     if dr>l then begin
20       inc(l); q[l+1]:=inf; end;
21     q[st]:=k; insert:=st;
22   end else
23     if k<q[m] then
24       insert:=insert(k,st,m)
25     else
26       insert:=insert(k,m+1,dr);
27 end;
28
29 procedure solve;
30 var i,k:integer;
31 begin
32   l:=0; q[1]:=inf;
33   for i:=1 to n do
34     p[i]:=insert(v[i],1,l+1);
35   k:=n;
36   for i:=l downto 1 do begin
37     while p[k]>>i do dec(k);
38     s[i]:=v[k];
39   end;
40   writeln(l);
41   for i:=1 to l do write(s[i],' ');
42   writeln;
43 end;
44 begin citire; solve end.

```

```

#include <stdio.h>
#define INF 0x3f3f
int N,L;
int V[10001],P[10001],
      Q[10001],S[10001];

void citire() {
  int i;
  freopen("subsr.in","r",stdin);
  freopen("subsr.out","w",stdout);
  scanf("%d",&N);
  for(i=1;i<=N;i++) scanf("%d",&v[i]);
}

int insert(int k,int st,int dr) {
  int m=(st+dr)/2;
  if (st==dr) {
    if (dr>L) Q[++L+1]=INF;
    Q[st]=k;
    return st;
  }
  if (k<Q[m]) return insert(k,st,m);
  return insert(k,m+1,dr);
}

void solve() {
  int i,k;
  L=0; Q[1]=INF;
  for (i=1;i<=N;i++)
    P[i]=insert(V[i],1,L+1);
  k=N;
  for (i=L;i;i--) {
    while (P[k]!=i) k--;
    S[i]=V[k];
  }
  printf("%d\n",L);
  for (i=1;i<=L;i++)
    printf("%d ", S[i]);
  printf("\n");
}

int main() {
  citire();
  solve();
  return 0;
}

```

**S. (\*\*\*)** În bucatăria unei gospodine se află un teanc format din  $N \leq 1.000$  farfurii. Gospodina care urmează să le spele este pusă în fața unei probleme. Ea cunoaște cât timp durează spălarea fiecărei farfurii. De asemenea, ea poate să renunțe la spălarea unei și să o spargă, însă spargerea durează exact un minut, pentru oricare farfurie. Farfurile sunt numerotate cu numere de la 1 la  $N$ , începând cu farfurie din vârful teancului spre cea de la baza teancului. Farfuria care poate fi spălată la un moment dat este cea situată în vârful teancului. Inițial gospodina poate spăla sau sparge farfurie cu numărul 1, după care poate face același lucru cu farfurie numărul 2 și a.m.d. De asemenea, gospodina poate lăsa în teanc farfurii nespălate. Este cunoscut timpul  $T \leq 10.000$  exprimat în minute, pe care gospodina îl are la dispoziție.

Realizați un program care determină care este numărul maxim de farfurii pe care le poate spăla gospodina în timpul  $T$  dat. Pentru acest număr maxim de farfurii obținut, determinați și timpul minim necesar spălării lor.

Fișierul de intrare *farfurii.in* conține pe prima linie două numere naturale  $M$  și  $N$  separate printr-un spațiu, reprezentând numărul de farfurii și respectiv timpul avut la dispoziție de gospodină. Pe următoarea linie se găsește un sir de  $N$  numere naturale  $ts_1, ts_2, \dots, ts_N$ , separate prin câte un spațiu, reprezentând timpul de spălare și fiercări farfurii, în ordine, începând cu farfuria 1.

În fișierul de ieșire *farfuri.out* se vor scrie pe o singură linie numerele *Nmax* și *Tmin* despărțite printr-un spațiu. *Nmax* reprezintă numărul maxim de farfuri pe care le poate spăla gospodina în timpul *T*, iar *Tmin*, timpul minim de spălare a lor.

### Exemplu:

[farfurii.in](http://farfurii.in)

4 4  
1 4 1 3

2

farfurii.out

Soluție:

#### Descompunerea problemei în subprobleme

Pentru șirul primelor  $k$  ( $k \leq n$ ) farfurii se determină care este numărul maxim de farfurii care pot fi spălate la minutul  $1, 2, \dots, T$ .

*Structuri de date utilizate și relațiile de recurență*

La finalul fiecărei iterări ( $k$ ) în vectorul  $B$ , elementul  $B[i]$  reține numărul maxim de farfurii, din sirul primelor  $k$ , care pot fi spălate în primele  $i$  minute.

Vectorul  $A$  retine configurația vectorului  $B$  la iteratia  $k-1$

Relația de recurență:  $B[i] = \max\{A[i-1], A[i-ts[k]] + 1\}$

```

1 var a,b:array [0..10200] of integer;
2 temp:array [1..10200] of integer;
3 maxim,n,t,i,k,max,tt:integer;
4 f:text;
5
6 procedure load;
7 begin
8   assign(f,'farfurii.in');
9   reset(f); readln(f,n,t);
10  for i:=1 to n do read(f,temp[i]);
11  close(f);
12 end;

```

```

void load() {
    f=fopen("farfurii.in","r");
    fscanf(f,"%d %d",&n,&t);
    for (i=1;i<=n;i++)
        fscanf(f,"%d",&tmp[i]);
    fclose(f);
}

void init() {
    a[0]=0;
    for (i=1;i<=t;i++) a[i]=-1;
}

```

```

13 procedure init;
14 begin a[0]:=0;
15   for i:=1 to t do
16     a[i]:=-maxint;
17   end;
18 procedure dinamica;
19 begin
20   maxim:=0; tt:=0;
21   for k:=1 to n do begin
22     for i:=0 to t do begin
23       max:=-maxint;
24       if i>=temp[k] then
25         if a[i-temp[k]]<>-maxint then
26           if a[i-temp[k]]+1>max then
27             max:=a[i-temp[k]]+1;
28       if i>=1 then
29         if a[i-1]<>-maxint then
30           if a[i-1]>max then max:=a[i-1];
31         b[i]:=max;
32       end;
33       for i:=0 to t do
34         if (b[i]>maxim)or((b[i]=maxim)
35           and (i<tt)) then begin
36           maxim:=b[i]; tt:=i; end;
37         for i:=0 to t do a[i]:=b[i];
38       end;
39     end;
40   procedure scrie;
41   begin
42     assign(f,'farfurii.out');
43     rewrite(f);
44     writeln(f,maxim,' ',tt);
45     close(f); end;
46   begin
47     load; init; dinamica; scrie;
48   end.

```

---

```

void dinamica() {
  maxim=tt=0;
  for (k=1;k<=n;k++) {
    for (i=0;i<=t;i++) {
      max=-32768;
      if (i>=temp[k])
        if (a[i-temp[k]]!= -32768)
          if (a[i-temp[k]]+1>max)
            max=a[i-temp[k]]+1;
      if (i>=1)
        if (a[i-1]== -32768)
          if (a[i-1]>max)
            max=a[i-1];
      b[i]=max;
    }
    for (i=0;i<=t;i++)
      if ((b[i]>maxim) || (b[i]==maxim)
          && i<tt))
        { maxim=b[i];
          tt=i; }
    for (i=0;i<=t;i++)
      a[i]=b[i];
  }
}

void scrie()
{f=fopen("farfurii.out","w");
printf("%d %d\n",maxim,tt);
fclose(f);
}

int main()
{load();
init();
dinamica();
scrie();
return 0; }

```

6. (\*\*\*\*\*) Datează o matrice binară de dimensiuni  $N \times M$  ( $3 \leq M, N \leq 1.000$ ) se cere să se determine aria dreptunghiului de întindere maximă care conține doar 1. Matricea se citește din fișierul *matrix.in* care va avea următoarea structură:

- pe prima linie cele două numere naturale  $N$  și  $M$ ;
  - următoarele  $N$  linii vor avea câte  $M$  cifre de 0 sau 1 despărțite printr-un spatiu.

Ceea ce se cere este să se scrie pe singura linie a fișierului *matrix.out* numărul natural care reprezintă aria determinată.

### *Exemplu*

```

matrix.i
8 9
1 1 0 1 1 1 0 1 1
0 0 1 0 1 1 0 0 0
1 0 0 1 0 1 0 1 0
0 0 1 0 1 1 1 1 0
1 0 0 1 1 0 1 1 0
1 1 1 1 0 1 1 1 1
0 1 1 1 1 1 1 1 1
1 1 0 1 1 1 1 1 1

```

matrix.out

三

### Soluție:

*Descompunerea problemei în subprobleme și structuri de date utilizate:*

Se va procesa matricea linie cu linie, iar pentru fiecare linie se va ține un vector  $h[1..m]$ ,  $h[i]=$ "înălțimea" coloanei  $i$ , adică numărul de linii strict consecutive care există mai sus de linia curentă (inclusiv) cu  $l$  pe coloana  $i$ .

Putem defini un dreptunghi maximal care pornește de la linia curentă și se aduce înapoi ca fiind dat de înălțimea unei coloane și având aria înălțime\*extindere.

Spre exemplu, pentru vectorul de înălțimi următor:

$i: \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8$

$h[i]: \quad 3 \ 3 \ 3 \ 2 \ 3 \ 3 \ 3 \ 3$

1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8

Coloana 1 are o înălțime de 3 și o extindere de 3 (poate "intra" în stânga și dreapta pe 3 coloane consecutive care sunt mai înalte sau egale cu ea). Deci coloana 1 dă un dreptunghi de  $3 \times 3 = 9$ , la fel și coloanele 2 și 3 care au aceeași înălțime și extindere. Coloana 4 are o înălțime de 2 și o extindere de 8 deoarece poate "intra" în toate coloanele, deci ne dă dreptunghi de  $2 \times 8 = 16$  (care este și maximal). Pentru coloanele 5, 6, 7, 8, avem înălțimea 3 și extinderea 4, deci dau un dreptunghi de 12. Extinderea se va calcula în 2 etape: extinderea spre dreapta și spre stânga.

Vom prezenta în continuare cum se calculează extinderea spre stânga (extinderea spre dreapta se calculează analog). Vectorul  $l$  pentru extinderea stânga ar trebui să arate astfel:  $l=(1,2,3,4,1,2,3)$ . Ca să se calculeze extinderea stânga liniar se va folosi o stivă, și se vor introduce toate coloanele pe rând în ea. În principal trebuie ca pentru fiecare coloană  $i$  să se știe care este prima coloană  $j$  (pornind de la  $i-1$  până la 1) care are o înălțime strict mai mică decât coloana  $i$ ; extinderea stânga va fi  $i-j$ .

Stiva se va ține cu un vector,  $q[i]=$ indicele unei coloane. Structura va fi tot timpul ordonată strict crescător după  $q$  și după  $h[q]$ , adică  $i < j \rightarrow q[i] < q[j]$  și  $h[q[i]] < h[q[j]]$ . Dacă suntem la coloana  $i$  și sunt deja  $k$  coloane în structură, pentru a introduce pe  $i$ , atunci cât timp înălțimea coloanei de pe poziția  $k$  în structură este mai mare sau egală cu înălțimea coloanei  $i$  se elimină elementul  $q[k]$  și se decrementează  $k$ ; coloana  $i$  va fi pusă la  $k+1$ , iar  $l[i]$  va fi  $i-q[k]$ .

Matricea e procesată pe linii, fiecare linie în complexitate liniară, deoarece în stivă se adaugă fiecare coloană o singură dată, iar fiecare coloană este scoasă cel mult odată.

```

1 var n,m,i,j,k,nn,max:longint;
2   l,h,q:array[0..1001] of longint;
3 begin
4   assign(input,'matrix.in');
5   reset(input);
6   readln(n,m);
7   max:=0;
8   #include <stdio.h>
9   #include <string.h>
10  long n,m,i,j,k,nn,max,
11    h[1001],q[1001],l[1001];
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37

```

```

8   for k:=1 to m do begin
9     for j:=1 to m do begin
10       read(i);
11       if i=1 then inc(h[j])
12         else h[j]:=0;
13     end;
14     fillchar(l,sizeof(l),0);
15     h[0]:=-1; q[0]:=0; nn:=0;
16     for i:=1 to m do begin
17       j:=nn;
18       while h[q[j]]>=h[i] do dec(j);
19       inc(j); nn:=j; inc(l[i],i-q[j-1]);
20       q[j]:=i;
21     end;
22     h[m+1]:=-1; q[0]:=m+1; nn:=0;
23     for i:=m downto 1 do begin
24       j:=nn;
25       while h[q[j]]>=h[i] do dec(j);
26       inc(j); nn:=j;
27       inc(l[i],q[j-1]-i-1); q[j]:=i;
28     end;
29     for i:=1 to m do
30       if max< l[i]*h[i] then
31         max:=l[i]*h[i];
32     end;
33     close(input);
34     assign('matrix.out',fo);
35     rewrite(output); write(max);
36     close(output);
37   end.

```

```

int main()
{
  freopen("matrix.in","r",stdin);
  scanf("%d %d",&n,&m);max=0;
  for (k=1;k<=n;k++)
  {
    for (j=1;j<=m;j++)
    {scanf("%d",&i);
     h[j]=i==1?h[j]+1:0;
    }
    memset(l,0,sizeof(l));
    h[0]=-1;
    q[0]=nn=0;
    for (i=1;i<=m;i++)
    {
      for (j=nn;h[q[j]]>=h[i];j--)
      nn++;j;l[i]+=i-q[j-1];q[j]=i;
    }
    for (i=1;i<=m;i++)
      if (max<l[i]*h[i]) max=l[i]*h[i];
  }
  freopen("matrix.out", "w", stdout);
  printf("%d\n",max);
  return 0;
}

```

7. (\*\*\*) Într-o companie cu  $N$  oameni, fiecare persoană este fie șef, fie angajat. Fiecare șef interacționează cu fiecare angajat din departamentul său (dar nu cu alți șefi). Gradul de eficiență al unui departament este egal cu numărul de interacțiuni distincte de tip șef – angajat, care au loc. Eficiența întregii companii este suma gradelor de eficiență al fiecărui departament. Se știe că eficiența companiei este  $E \leq 2.000$ , trebuie să se determine numărul minim de angajați, dintre care să fie un număr minim de șefi, pentru a se obține această eficiență.

Prima linie a fișierului *comp.in* conține numărul întreg  $E$ .

Prima linie a fișierului *comp.out* conține numărul minim de oameni din companie, numărul minim de șefi.

*Exemplu:*

<i>comp.in</i>		<i>comp.out</i>
7		7 3

### Soluție:

*Descompunerea problemei în subprobleme și structuri de date utilizate:*

Se vor construi doi vectori:

$P[i]$  = numărul minim de persoane necesare pentru a obține eficiență  $i$ ;

$S[i]$  = numărul minim de șefi necesari pentru a obține eficiență  $i$ .

### Relațiile de recurență:

Se iau toate grupurile posibile de  $x$  persoane, dintre care se actualizează valorile  $P[i+(x-y)*y]$  și  $S[i+(x-y)*y]$ , în funcție de  $P[i]$  și  $S[i]$ .  
 $\{P[i+(x-y)*y], S[i+(x-y)*y]\} = \min\{P[i+(x-y)*y], S[i+(x-y)*y]\}, \{P[i]+x, S[i]+y\}$

```

1 const inf=16191;
2 var e,i,j,a,b,x,y:integer;
3 p,s:array[0..2000] of integer;
4 begin
5 assign(input,'comp.in');
6 reset(input);
7 assign(output,'comp.out');
8 rewrite(output); readin(e);
9 for i:=1 to e do begin
10   p[i]:=inf; s[i]:=inf;
11 end;
12 for i:=0 to e do
13   for x:=1 to e do begin
14     y:=1;
15     while(y<x)and(i+(x-y)*y<=e)do
16       begin
17         a:=p[i]+x; b:=s[i]+y;
18         j:=i+(x-y)*y;
19         if (p[j]>a)or((p[j]=a)and
20           (s[j]>b)) then begin
21           p[j]:=a; s[j]:=b;
22         end;
23         inc(y);
24       end;
25   end;
26   writeln(p[e],' ',s[e]);
27 end.

```

```

#include <stdio.h>
#define INF 0x3f3f

int E,P[2001],S[2001];

int main()
{
    int i,j,x,y,a,b;
    freopen("comp.in","r",stdin);
    freopen("comp.out","w",stdout);
    scanf("%d",&E);
    for(i=1;i<=E;i++)
        P[i]=S[i]=INF;
    for(i=0;i<=E;i++)
        for(x=1;x<=E;x++)
            for(y=1;y<=i+(x-y)*y<=E;y++)
            {
                a=P[i]+x; b=S[i]+y;
                j=i+(x-y)*y;
                if ((p[j]>a) || (p[j]==a&&s[j]>b))
                {
                    p[j]=a;
                    s[j]=b;
                }
            }
    printf("%d %d\n",P[E],S[E]);
    return 0;
}

```

8. (\*\*\*) Fermierul Ion are o fermă de formă circulară, unde cresc  $N \leq 10.000$  găini. Ferma a fost împărțită în  $N$  sectoare, numerotate de la 1 la  $N$ , astfel încât oricare două sectoare având numere consecutive sunt adjacente (se află unul lângă altul). În plus, primul și ultimul sector sunt adjacente. În fiecare sector se află câte o găină, iar aceasta depune un anumit număr de ouă în fiecare zi. După ce găinile depun ouăle, fermierul Ion dorește să le adune, pentru a le mâncă. Deoarece fermierul este foarte lacom, de fiecare dată el alege două sectoare adjacente din care adună ouăle simultan. Din păcate, din cauza lăcomiei sale, găinile din sectoarele vecine cu cele doă alese se sperie și devin violente, motiv pentru care fermierul nu mai poate aduna ouăle din aceste sectoare. Determinați numărul maxim de ouă pe care le poate aduna fermierul Ion, în urma aplicării strategiei sale lacome.

Fișierul de intrare *oo.in* conține pe prima linie numărul de sectoare în care este împărțită ferma (și, implicit, numărul de găini). Pe următoarea linie se află  $N$  numere întregi din intervalul  $[0,100]$ , reprezentând numărul de ouă depuse de fiecare găină, în ordinea sectoarelor în care se află acestea.

În fișierul *oo.out* veți afișa numărul maxim de ouă pe care le poate aduna fermierul Ion.

### Exemplu:

<pre> 5 6 7 1 2 1 </pre>	<pre> oo.in 8 </pre>	<pre> oo.out </pre>
--------------------------	----------------------	---------------------

(Concursul național "Steile informaticii" 2003, București)

### Soluție:

*Descompunerea problemei în subprobleme și structuri de date utilizate:*

Se construiesc trei tablouri unidimensionale (conceptual; practic, se va folosi unul singur):

- $A_1$  – elementul de pe poziția  $i$  reprezintă numărul maxim de ouă adunate până în sectorul  $i$ , dacă se adună ouăle din primele două sectoare, luate împreună;
- $A_2$  – elementul de pe poziția  $i$  reprezintă numărul maxim de ouă adunate până în sectorul  $i$ , dacă se adună ouăle din sectorul al doilea, împreună cu cele din al treilea;
- $A_3$  – elementul de pe poziția  $i$  reprezintă numărul maxim de ouă adunate până în sectorul  $i$ , dacă se adună ouăle din primul și ultimul sector, luate împreună.

Evident, în primul caz nu se pot lua ouăle din ultimul sector etc. Valorile din aceste tablouri se pot calcula pe baza unor recurențe simple, ceea ce conduce la un algoritm liniar.

În final se selectează maximul dintre aceste valori, determinate corect.

```

1 var n,i,j,k:integer;
2 rez:longint;
3 oo:array[0..9999] of integer;
4 a:array[0..9999] of longint;
5
6 function max(a,b:longint):longint;
7 begin if a>b then max:=a else max:=b;
8 end;
9
10 begin
11   ....
12   read(n);
13   for i:=0 to n-1 do read(oo[i]);
14   if n=2 then rez:=oo[0]+oo[1]
15   else
16     for k:=0 to 1 do begin
17       a[k]:=0; a[(k+1)mod n]:=0;
18       a[(k+2)mod n]:=oo[(k+1)mod n]+
19                     oo[(k+2)mod n];
20       i:=(k+3) mod n;
21       while i>k do begin
22         a[i]:=max(a[(i-1+n)mod n],
23                    a[(i-3+n)mod n]+
24                    oo[(i-1+n)mod n]+oo[i]);
25         i:=(i+1) mod n;
26       end;
27       rez:=max(rez,a[(k-1+n)mod n]);
28     end;
29   writeln(rez); end.

```

```

#include <stdio.h>
int n,oo[10000];
long A[10000],rez;

long max(long a, long b)
{ return a > b ? a : b; }

void main()
{
    int i,j,k;
    freopen("oo.in","r",stdin);
    scanf("%d", &n);
    for (i=0;i<n;i++) scanf("%d",oo+i);
    if (n==2) rez=oo[0]+oo[1];
    else
        for (k=0; k<2; k=(k+1)%n)
    {
        A[k]=0; A[(k+1)%n]=0;
        A[(k+2)%n]=oo[(k+1)%n]+oo[(k+2)%n];
        for (i=(k+3)%n; i!=k; i=(i+1)%n)
            A[i]=max(A[(i-1+n)%n],
                      A[(i-3+n)%n]+
                      oo[(i-1+n)%n]+oo[i]);
        rez=max(rez, A[(k-1+n)%n]);
    }
    freopen("oo.out","w",stdout);
    printf("%ld\n", rez);
}

```

9. (\*\*\*) În fiecare zi, Zăhărel este obligat de Eugenia să spele farfuriiile și tacâmurile după fiecare masă. După ce le spălă, trebuie să le aranjeze pe două rafturi, farfuriiile pe primul și tacâmurile pe al doilea ... dar nu oricum! El are  $N \leq 10.000$  farfurii de mărime distincte, cuprinse între 1 și  $N$  și  $K \leq N^*(N-1)/2$  tacâmi identice. Pentru fiecare pereche de farfurii așezate în raft astfel încât farfuria de mărime mai mare, dintre cele două, să apară înaintea farfuriei de mărime mai mică, Zăhărel pune un tacâm pe rândul al doilea.

Ajutați-l pe Zăhărel să așzeze toate farfuriiile pe primul raft astfel încât să pună toate tacâmurile pe al doilea raft. Dintre toate așezările posibile determinați-o pe aceea minim lexicografică din punct de vedere al mărimilor. O așezare  $(A_1, A_2, \dots, A_N)$  este mai mică din punct de vedere lexicografic decât o altă așezare  $(B_1, B_2, \dots, B_N)$  dacă există o poziție  $p$  astfel încât  $A_p < B_p$  și  $A_1 = B_1, A_2 = B_2, \dots, A_{p-1} = B_{p-1}$ .

Pe prima linie din fișierul de intrare *farfurii.in* se găsesc numerele naturale  $N$  și  $K$ . Pe prima linie din fișierul de ieșire *farfurii.out* se vor găsi  $N$  numere distincte între 1 și  $N$  reprezentând mărimile farfuriiilor, afișate în ordinea în care au fost așezate pe raft.

*Exemplu:*

*farfurii.in*

7 8

*farfurii.out*

1 2 5 7 6 4 3

*Soluție:*

O rezolvare *greedy* simplă se bazează pe următoarea observație: o permutare de lungime  $i$  are cel mult  $i*(i-1)/2$  inversiuni când numerele sunt în ordine descrescătoare.

Astfel, dacă  $K$  e de forma  $M*(M-1)/2$ , permutarea minim lexicografică cu  $K$  inversiuni va fi  $1, 2, 3, \dots, N-M, N, N-1, N-2, \dots, N-M+1$ . Cele  $K$  inversiuni apar în ultimele  $M$  elemente. Dacă în această permutare se mută un element  $N-x$  imediat înaintea lui  $N$ , numărul de inversiuni scade cu  $x$ . Astfel, dacă  $K > M*(M-1)/2$  se construiește permutarea:

1, 2, 3, ...  $N-M-1, N, N-1, N-2, \dots, N-M$ .

(care are  $(M+1)*M/2$  inversiuni) și se mută elementul  $N - ((M+1)*M/2 - K)$  imediat înaintea lui  $N$ , astfel se scade numărul de inversiuni la  $K$ . Este evident că permutarea astfel construită este minim lexicografică.

```

1 var n,k,i,m,p:longint;
2 begin
3   assign(input,'farfurii.in');
4   reset(input);
5   assign(output,'farfurii.out');
6   rewrite(output); readln(n,k);
7   p:=1; while p<n do p:=p*2;
8   m:=0;
9   while p>0 do begin
10     if (m+p<=n)and
11     ((m+p)*(m+p-1)<=2*k)then m:=m+p;
12     p:=p div 2; end;
13   writeln(m);
14   if k=0 then begin
15     for i:=1 to n-m do write(i,' ');
16     for i:=n downto n-m+1 do
17       write(i,' ');
18     writeln; end
19   else begin
20     k:=m-k; inc(m);
21     for i:=1 to n-m do write(i,' ');
22     write(n-k,' ');
23     for i:=n downto n-k+1 do
24       write(i,' ');
25     for i:=n-k+1 down to n-m+1 do
26       write(i,' ');
27     writeln;
28   end
29 end.

```

```

if (1k) {
  for (i=1;i<=n-m;i++)
    printf("%ld ",i);
  for (i=n;i>n-m;i--)
    printf("%ld ",i);
  putchar ('\n');
}
else {
  k=m-k; m++;
  for (i=1;i<=n-m;i++)
    printf("%ld ",i);
  printf ("%ld ",n-k);
  for (i=n;i>n-k;i--)
    printf("%ld ",i);
  for (i=n-k+1;i>n-m;i--)
    printf("%ld ",i);
  putchar ('\n');
}

```

10. Înainte să se apuce de informatică, Bronzărel avea altă ocupație, și anume era negustor de animale. Fiindcă a renunțat la această profesie pentru cea de informatician, trebuie acum să-și vândă animalele. La târg, el nu poate vinde doar un animal, ci trebuie să le vândă pe grupuri, fiecare grup fiind format din fix  $K$  animale de tipuri distincte. Știind că Bronzărel avea  $N \leq 10.000$  tipuri de animale și cantitatea  $A_i \leq 10.000$  din fiecare tip, determinați care este numărul maxim de grupuri pe care le poate forma, pentru a le vinde la târg.

Prima linie a fișierului *grupuri.in* va conține numerele naturale  $K$  și  $N$ . Următoarea linie va conține  $N$  numere naturale  $A_1, A_2, \dots, A_N$  reprezentând cantitățile disponibile din fiecare tip de animal. Cantitățile vor fi date în ordine crescătoare ( $A_i \leq A_{i+1}$ ).

Pe prima linie din fișierul *grupuri.out* se va scrie o singură valoare reprezentând numărul maxim de grupuri care se pot forma.

*Exemplu:* *grupuri.in*

3 4  
3 3 3 3

*grupuri.out*

4

*Soluție:*

O limită superioară pentru numărul de grupuri este  $S / K$  unde  $S$  reprezintă suma cantităților de animale. Este evident că, dacă putem construi  $X$  grupuri, putem construi și  $Y \leq X$  grupuri, fapt care duce la ideea să căutăm binar numărul de grupuri. Pentru a verifica dacă se poate construi un număr de grupuri  $X$  vom considera completarea unei matrice imaginare în mod *greedy* cu  $X$  linii și  $K$  coloane, fiecare linie reprezentând un grup. Din restricțiile problemei elementele de pe fiecare linie trebuie să fie distincte. Se va completa această matrice coloană cu coloană, pentru fiecare cantitate de animale, dacă este  $\leq X$  se pun toate în matrice, dacă este  $> X$  se pun doar  $X$  în matrice (deoarece dacă se pun mai multe vor fi cel puțin două pe aceeași linie). Pentru primul exemplu din enunț, matricea se va completa astfel:

1 * *	1 2 *	1 2 3	1 2 3
1 * *	1 2 *	1 2 *	1 2 4
1 * *	1 * *	1 3 *	1 3 4
* * *	2 * *	2 3 *	2 3 4
=>	=>	=>	=>

Așadar, dacă în final s-a completat toată matricea, se pot forma  $K$  grupuri. Completarea matricei se face doar conceptual, verificarea având complexitatea  $O(N)$  prin parcurgerea vectorului  $A$ , rezultând un algoritm de complexitate  $O(N \lg(S/K))$ . Pe baza algoritmului descris mai sus, se poate obține un algoritm  $O(N)$ , astfel: dacă cel mai mare element este  $\leq S/K$ , verificarea prezentată mai sus garantează că se vor putea obține  $S/K$  grupuri, toate elementele fiind  $\leq S/K$  se vor folosi toate  $S$ , iar matricea este  $(S/K) * K = S$ . Dacă cel mai mare element este  $> S/K$ , atunci se poate rezolva aceeași problemă recursiv pentru cantitățile existente și mai puțin cea mai mare și grupuri de mărime  $K-1$ . Acest rezultat va fi mai mic sau egal cu cel mai mare element, deci va ajunge pentru a completa grupurile la mărimea  $K$ . Deoarece cantitățile sunt date sortate, complexitatea este  $O(N)$ .

```

1 var n,k,i:integer; sum:longint;
2 a:array[0..10000] of integer;
3
4 function solve(n:integer;sum:longint;
5 k:integer):longint;
6 begin
7   if (n<0)or(k<=0) then begin
8     solve:=0; exit; end;
9   if a[n]<=sum div k then
10    solve:=sum div k
11   else
12    solve:=solve(n-1,sum-a[n],k-1);
13  end;
14
15 begin
16  assign(input,'grupuri.in');
17  reset(input);
18  assign(output,'grupuri.out');
19  rewrite(output);
20  readln(k,n);
21  for i:=0 to n-1 do begin
22    read(a[i]); inc(sum,a[i]);
23  end;
24  writeln(solve(n-1,sum,k));
25 end.

```

```

#include <stdio.h>
#include <math.h>

int N, K, A[10001];
long solve(int n, long sum, int k) {
  if (n<0 || k<=0) return 0;
  if (A[n]<=sum/(long)k)
    return sum/(long)k;
  return solve(n-1, sum-A[n], k-1);
}

int main() {
  int i;
  long sum = 0;
  freopen("grupuri.in", "r", stdin);
  freopen("grupuri.out", "w", stdout);
  scanf("%d %d", &K, &N);
  for (i=0; i<N; i++) {
    scanf("%d", &A[i]);
    sum += (long) A[i];
  }
  printf("%ld\n", solve(N-1, sum, K));
  return 0;
}

```

11. (\*\*\*\*) Zăhărel are un interval  $[1..L]$  ( $L \leq 1.000.000.000$ ) și  $N \leq 10.000$  alte intervale incluse în intervalul  $[1..L]$ . El poate atribui fiecărui din cele  $N$  intervale o culoare, alb sau negru. Ajutați-l să determine o astfel de colorare cu proprietatea că, atât reuniunea intervalelor de culoare albă să fie intervalul  $[1..L]$  cât și reuniunea intervalelor de culoare neagră să fie tot intervalul  $[1..L]$ .

Fișierul de intrare *int.in* conține pe prima linie numerele  $L$  și  $N$ , cu semnificația din enunț. Pe următoarele  $N$  linii se găsesc perechi de numere întregi reprezentând intervalele.

Fișierul de ieșire *int.out* trebuie să conțină  $N$  linii, pe fiecare se va afla căte o cifră, 0 sau 1 (0 pentru negru și 1 pentru alb), linia  $i$  din fișierul de ieșire reprezentând culoarea intervalului de pe linia  $i+1$  din fișierul de intrare. Dacă nu este posibilă o astfel de colorare se va afișa doar mesajul "Nu are soluție".

### Exemplu:

*int.in*

```

7 4
1 3
1 5
5 7
3 7

```

*int.out*

```

1
0
0
1
1

```

### Soluție:

Pentru a rezolva această problemă se va aplica o strategie greedy: se sortează intervalele după capătul stânga și se atrbuie o culoare intervalului în funcție de ce culoare este cea mai puțin "acoperită".

La sfârșit, se verifică dacă ambele culori acoperă întregul interval  $[1..L]$ .

```

1 var n,l,i,w,b:longint;
2 a:array[0..10000,0..2]of longint;
3 c:array[0..10000] of byte;
4
5 procedure sort;
6 begin
7   ...
8 end;
9
10 begin
11  assign(input,'int.in');
12  reset(input);
13  assign(output,'int.out');
14  rewrite(output);
15  readln(l,n);
16  for i:=0 to n-1 do begin
17    read(a[i,0],a[i,1]);
18    a[i,2]:=i;
19  end;
20  sort; w:=1;b:=1;
21  for i:=0 to n do
22    if w<b then begin
23      if a[i,0]>w then break;
24      if w<a[i,1] then w:=a[i,1];
25      c[a[i,2]]:=0;
26    end else begin
27      if a[i,0]>b then break;
28      if b<a[i,1] then b:=a[i,1];
29      c[a[i,2]]:=1;
30    end;
31  if (i<n)or(w<l)or(b>l) then
32    writeln('Nu are solutie');
33  else
34    for i:=0 to n-1 do
35      writeln(c[i]);
36    close(output);
37 end.

```

```

#include <stdio.h>
#include <stdlib.h>

long N, L, A[10001][3];
char C[10001];

int cmp(const void *i,const void *j){
  ...
}

int main() {
  long i, w, b;
  freopen("int.in", "r", stdin);
  freopen("int.out", "w", stdout);
  scanf("%ld %ld", &L, &N);
  for (i=0; i<N; i++) {
    scanf("%ld %ld", &A[i][0], &A[i][1]);
    A[i][2] = i;
  }
  qsort(A, N, sizeof(long)*3, cmp);
  for (w=b=1, i=0; i<N; i++) {
    if (w < b) {
      if (A[i][0] > w) break;
      if w<a[i,1] then w:=a[i,1];
      C[A[i][2]] = 0;
    } else {
      if (A[i][0] > b) break;
      if b<a[i,1] then b:=a[i,1];
      C[A[i][2]] = 1;
    }
    if (i < N || w < L || b < L)
      printf("Nu are solutie\n");
    else
      for (i = 0; i < N; i++)
        printf("%d\n", C[i]);
    return 0;
}

```

12. (\*\*\*) Zăhărel se joacă adesea cu șiruri de paranteze (evident că folosește doar șiruri corecte!). Uneori ajunge să aibă secvențe mari de tipul "))))...)"'. Pentru a preveni astfel de situații a inventat "paranteza magică ]" care înlocuiește una sau mai multe paranteze ")", după cum este nevoie pentru a închide corect parantezele deschise. Determinați, pentru un șir dat, câte paranteze închise înlocuiește.

Fisierul de intrare *parantez.in* conține două linii; prima linie conține lungimea șirului, iar cea de a doua, șirul.

Fisierul de ieșire *parantez.out* conține atâtea linii câte paranteze magice sunt în șirul dat. Pe fiecare linie se va scrie câte paranteze închise înlocuiește fiecare paranteză magică, luate în ordinea în care apar în șirul dat de la stânga la dreapta.

*Exemplu:*

<i>parantez.in</i>	<i>parantez.out</i>
8 (((())))	3 1

*Soluție:*

Se aplică o strategie *greedy* pentru a rezolva această problemă: fiecare paranteză magică, mai puțin ultima, este înlocuită cu o singură paranteză normală (")").

Pentru ultima paranteză magică se introduc câte paranteze normale sunt necesare pentru a forma o expresie corectă.

```

1 var n,i,j,nr:integer;
2   s:string;
3 begin
4   assign(input,'parantez.in');
5   reset(input);
6   assign(output,'parantez.out');
7   rewrite(output);
8   readln(n);
9   readln(s);
10  for i:=n downto 1 do
11    if s[i]=']' then break;
12  for j:=1 to i-1 do
13    if s[j]='[' then begin
14      writeln(1); dec(nr);
15    end else
16      if s[j]='(' then inc(nr)
17    else
18      if s[j]=')' then dec(nr);
19    writeln(nr);
20  end.

```

```

#include <stdio.h>
#include <string.h>
int N; char S[10001];

int main() {
    int i,j,nr=0;
    freopen("parantez.in","r",stdin);
    freopen("parantez.out","w",stdout);
    scanf("%d\n%s\n", &N,S);
    for (i=N-1;i>=0;i--)
        if (S[i]==']') break;
    for (j=0;j<i;j++) {
        if (S[j]=='[') {
            printf("1\n");
            nr--;
        }
        else if (S[j]==('(')) nr++;
        else if (S[j]=='))') nr--;
    }
    printf("%d\n",nr); return 0;
}

```

13. (\*\*\*) Gigel a descoperit un nou joc pe care l-a numit „*Flip*”. Acesta se joacă pe o tablă dreptunghiulară de dimensiuni  $N \times M$  ( $1 \leq N, M \leq 16$ ) care conține numere întregi. Fiecare linie și fiecare coloană are un comutator care schimbă starea tuturor elementelor de pe acea linie sau coloană, înmulțindu-le cu -1. Scopul jocului este ca pentru o configurație dată a tablei de joc să se acționeze asupra liniilor și coloanelor astfel încât să se obțină o tablă cu suma elementelor cât mai mare.

Prima linie a fișierului *flip.in* conține două numere întregi  $N$  și  $M$ , separate prin căte un spațiu, care reprezintă dimensiunea tablei. Următoarele  $N$  linii conțin căte  $M$  numere întregi, separate prin căte un spațiu care descriu configurația tablei de joc. Tabla de joc conține numere întregi din intervalul  $[-1.000.000, 1.000.000]$ .

Prima linie a fișierului *flip.out* conține un număr care va reprezenta suma maximă pe care Gigel o poate obține comutând linile și coloanele tablei de joc.

*Exemplu:*

<i>flip.in</i>	<i>flip.out</i>
15 3 4 -2 2 3 -1 5 2 0 -3 4 1 -3 5 -3 2	28

*Solutie:*

Soluția constă în a genera, folosind *backtracking*, acțiunile de comutare de pe prima linie a tabloului. În acest mod, pe fiecare linie se va alege dacă se comută sau nu linia, în funcție de modul în care se obține sumă mai mare. Dintre toate variantele de acțiuni, se va păstra aceea cu suma maximă.

```

1 var n,m,smax,i,j:longint;
2   a:array[0..15,0..15] of longint;
3   st:array[0..15] of longint;
4   procedure back(k:integer);
5   var i,j,s,t:integer;
6   begin
7     if k=m then begin
8       s:=0;
9       for i:=0 to n-1 do begin
10         t:=0;
11         for j:=0 to m-1 do
12           if (st[j]=1) then inc(t,-a[i][j])
13             else inc(t,a[i][j]);
14         if t>-t then inc(s,t)
15           else inc(s,-t);
16       end;
17       if smax<s then smax:=s;
18       exit;
19     end;
20     st[k]:=0; back(k+1);
21     st[k]:=1; back(k+1);
22   end;
23 begin
24   assign(input,'flip.in');
25   reset(input); readln(n,m);
26   for i:=0 to n-1 do
27     for j:=0 to m-1 do
28       read(a[i,j]);
29     back(0);
30   assign(output,'flip.out');
31   rewrite(output); write(smax);
32 end.

```

```

#include <stdio.h>
long n,m,a[16][16],
      st[16],smax;
void back(int k) {
    int i,j,s,t;
    if (k==m) {
        s=0;
        for (i=0;i<n;i++) {
            for (j=0;j<m;j++)
                if (st[j]) t+=-a[i][j];
                else t+=a[i][j];
            s+=t<-t ? -t : t;
        }
        if (smax<s) smax=s;
        return;
    }
    st[k]=0; back(k+1);
    st[k]=1; back(k+1);
}
int main()
{
    long i,j;
    freopen("flip.in","r",stdin);
    scanf("%ld %ld",&n,&m);
    for (i=0;i<n;i++)
        for (j=0;j<m;j++)
            scanf("%ld",&a[i][j]);
    back(0);
    freopen("flip.out","w",stdout);
    printf("%ld\n",smax);
    return 0;
}

```

34. (\*\*\*\*\*) Se consideră un sir de  $N \leq 10.000$  numere naturale  $t_1, t_2, \dots, t_N$  ( $0 \leq t_i \leq 20.000$ ). Să se determine o altă secvență de numere naturaleordonate crescător  $z_1 < z_2 < \dots < z_N$  astfel încât valoarea  $|t_1 - z_1| + |t_2 - z_2| + \dots + |t_N - z_N|$  să fie minimă.

Din fișierul *secv.in* se va citi valoarea  $N$  și apoi valorile  $t_1, t_2, \dots, t_N$ . Costul minim, căt și secvența  $z_1, z_2, \dots, z_N$  vor fi afișate în *secv.out*.

*Exemplu:*

*secv.in*

7  
9 4 8 20 14 15 18

*secv.out*

13  
6 7 8 13 14 15 18

*Soluție:*

În primul rând se va modifica problema inițială la cea de a determina un sir crescător (nu neapărat strict crescător) construind sirurile  $z'[i] = z[i] - i$  și  $t'[i] = t[i] - i$ . Astfel, o soluție  $z'$  pentru problema modificată este și o soluție pentru problema originală.

Pentru a rezolva noua problemă se va folosi *divide et impera*. Se va construi o procedură *solve(i, j, lo, hi)* care construiește valorile  $z[i], z[i+1], \dots, z[j]$  dându-le valori din intervalul  $[lo..hi]$ . Această procedură lucrează astfel:

- se determină, în timp liniar, cea mai mică poziție  $i \leq j$  cu proprietatea că  $z[p] > (lo+hi)/2$ ; se poate demonstra că această poziție este cea care respectă condiția ca pentru fiecare  $p \leq q \leq j$ , mediana numerelor  $t'[p], t'[p+1], \dots, t'[q]$  este mai mare decât  $(lo+hi)/2$ ;
- se apelează recursiv *solve(i, p, lo, (lo+hi)/2)* și *solve(p+1, j, (lo+hi)/2+1, hi)*.

Cazurile de bază când  $i = j$  sau  $lo = hi$  se rezolvă trivial.

```

1 var n,i,res:integer;
2 t,z:array[0..10000] of integer;
3
4 procedure solve(i,j,lo,hi:integer);
5 var k,m,nr,poz:integer;
6 begin
7   if i=j then begin
8     if (lo<=t[i])and(t[i]<=hi) then
9       z[i]:=t[i]
10    else
11      if t[i]<lo then z[i]:=lo
12      else z[i]:=hi;
13      inc(res,abs(z[i]-t[i]));
14      exit;
15   end;
16   if lo=hi then begin
17     for k:=i to j do begin
18       z[k]:=lo;
19       inc(res,abs(z[k]-t[k]));
20     end;
21     exit;
22   end;
23   m:=(lo+hi) div 2;

```

```

#include <stdio.h>
#define abs(x) ((x)<0?-(x):(x))

int N,T[10001],Z[10001],Res;

void solve(int i,int j,int lo,int hi)
{ int k, m, nr, poz;
  if (i == j) {
    if (lo<=T[i]&&T[i]<=hi)
      Z[i]=T[i];
    else
      if (T[i]<lo) Z[i]=lo;
      else Z[i]=hi;
    Res += abs(T[i]-Z[i]);
    return;
  }
  if (lo == hi){
    for (k=i; k<=j; k++) {
      Z[k]=lo;
      Res += abs(T[k]-lo);
    }
    return;
  }
}

```

```

24   nr:=0;
25   poz:=j+1;
26   for k:=j downto i do begin
27     if t[k]>m then inc(nr);
28     if t[k]<=m then
29       if nr>=0 then nr:=-1
30       else nr:=nr-1;
31     if nr>=0 then poz:=k;
32   end;
33   if i<=poz-1 then
34     solve(i,poz-1,lo,m);
35   if (poz<=j)and(m+1<=hi) then
36     solve(poz,j,m+1,hi);
37 end;
38
39 begin
40   assign(input,'secv.in');
41   reset(input);
42   assign(output,'secv.out');
43   rewrite(output);
44   readln(n);
45   for i:=0 to n-1 do begin
46     read(t[i]); dec(t[i],i);
47   end;
48   solve(0,n-1,0,20000);
49   writeln(res);
50   for i:=0 to n-1 do
51     write(z[i]+i,' ');
52   writeln;
53 end.

```

```

m=(lo+hi)/2;
nr=0;
poz=j+1;
for (k=j; k>=i; k--) {
  if (T[k]>m) nr++;
  if (T[k]<=m)
    nr = nr>=0 ? -1 : nr-1;
  if (nr >= 0) poz = k;
}

if (i<=poz-1) solve(i,poz-1,lo,m);
if (poz<=j && m+1<=hi)
  solve(poz,j,m+1,hi);
}

```

```

int main()
{int i;
freopen("secv.in","r",stdin);
freopen("secv.out","w",stdout);
scanf("%d", &N);
for (i = 0; i < N; i++)
  scanf("%d", &T[i]);
T[i] -= i;
}

solve(0, N-1, 0, 20000);
printf("%d\n", Res);
for (i = 0; i < N; i++)
  printf("%d ", Z[i]+i);
printf("\n");
return 0;
}

```

### 3.5.2 Probleme propuse

1. (\*\*\*\*\*) *Zăhărel* încearcă să învețe pe prietena lui Eugenia informatică. Astăzi a învățat-o programare dinamică și anume a început cu problema celui mai lung subșir comun: dându-se două siruri de lungime maxim 500, formate doar din litere mici, să se determine cel mai lung subșir comun al celor două siruri. Un subșir al unui sir este format din caractere (nu neapărat consecutive) ale sirului respectiv, în ordinea în care acestea apar în sir. Eugenia a înțeles rezolvarea problemei dar i-a pus următoarea întrebare lui *Zăhărel*: câte subșiruri comune de lungime maximă distincte există pentru cele două siruri? Două subșiruri sunt distincte dacă există cel puțin un caracter în unul din ele care diferă de caracterul din celălalt subșir de pe aceeași poziție.

Ajutați-l pe *Zăhărel* și determinați restul împărțirii numărului de subșiruri comune de lungime maximă distincte pentru două siruri date, la numărul 666013.

Pe prima linie a fișierului de intrare *subsir.in* se găsește primul sir, iar pe a doua linie, cel de-al doilea sir. Pe prima linie a fișierului de ieșire *subsir.out* se va găsi numărul cerut.

*Exemplu:*

*subsir.in*

banana  
oana

*subsir.out*

1  
(<http://infoarena.devnet.ro>)

2. (\*\*\*) *Gigel* s-a decis să devină olimpic la informatică, poate că și va reuși să-și rezolve singur problemele și nu va mai cere ajutorul vostru! La ora de informatică, profesorul lui i-a dat să rezolve problema secvenței de sumă maximă: „*Gigelu*, eu îți dau un sir de  $N \leq 50.000$  numere întregi din intervalul  $[-25.000, 25.000]$ , iar tu trebuie să găsești o secvență (adică un subșir de numere care apar pe poziții consecutive în sirul inițial) cu suma elementelor maximă!”. După vreo 30 de minute, *Gigel* s-a ridicat mândru și a zis: „Am găsit algoritmul de complexitate optimă, doamna profesoră!”. Ca temă pentru acasă, *Gigel* are de rezolvat aproape aceeași problemă: trebuie să găsească secvența de sumă maximă de lungime cel puțin  $K$ !

*Gigel* încă nu știe destul de multă informatică ca să poată rezolva această problemă, dar poate îl ajutați voi! Scrieți un program care rezolvă problema din tema lui *Gigel*.

Fișierul de intrare *secv2.in* conține pe prima linie numerele  $N$  și  $K$ , separate prin spațiu. Pe cea de a doua linie se află elementele sirului separate prin câte un spațiu. Fișierul de ieșire *secv2.out* trebuie să conțină o singură linie cu trei numere: poziția de început și de sfârșit a secvenței de sumă maximă de lungime cel puțin  $K$  și suma secvenței.

*Exemplu:*

secv2.in	secv2.out
8 0 -6 2 1 4 -1 3 -5	3 7 9

3. (\*\*\*\*\*+) *Gigel* are un sir de  $N \leq 500.000$  numere întregi din intervalul  $[-30.000, 30.000]$ . Toată lumea știe că o secvență este un subșir de numere care apar pe poziții consecutive în sirul inițial. *Gigel* a definit baza unei secvențe ca fiind minimul valorilor elementelor din secvență respectivă. Fiind dat un număr natural  $K$ , determinați pentru *Gigel* o secvență de lungime cel puțin  $K$  cu baza maximă. Fișierul de intrare *secventa.in* conține pe prima linie, numerele  $N$  și  $K$ , separate prin spațiu. Pe cea de a doua linie se află elementele sirului separate prin câte un spațiu.

Fișierul de ieșire *secventa.out* trebuie să conțină o singură linie cu trei numere: poziția de început și de sfârșit a secvenței de lungime cel puțin  $K$  cu baza maximă și valoarea maximă a bazei.

*Exemplu:*

secventa.in	secventa.out
8 3 -1 2 3 1 0 4 8 6	6 8 4

(<http://infoarena.devnet.ro>)

4. (\*\*\*\*\*+) *Gicu* și *Nicu*, olimpii la informatică și buni prieteni, mereu încearcă să îmbine activitățile lor cu informatică. Spre exemplu, când se plătăsc la ore, ei joacă un joc, bazat pe următoarele reguli:

- fie o matrice cu numere întregi cuprinse în intervalul  $[-1.000, 1.000]$ , cu  $N$  linii și  $M$  coloane ( $N, M \leq 1.000$ );

- liniile sunt numerotate de la 1 la  $N$ , iar coloanele de la 1 la  $M$ ;
- fiecare jucător mută alternativ un jeton plasat pe un element din matrice;
- o mutare constă în plasarea jetonului pe o altă poziție și adăugarea valorii din matrice de pe poziția respectivă la scorul jucătorului care a făcut mutarea; odată plasat jetonul pe o poziție, jucătorul următor poate să mute jetonul doar pe o altă poziție din dreptunghiul format de colțul stânga-sus și poziția curentă a matricei;
- jocul se termină când un jucător ajunge cu jetonul în colțul stânga-sus al matricei;
- la începutul jocului, ambiții jucători au scor 0, iar jucătorul care începe alege poziția inițială a jetonului.

Presupunând că fiecare din cei doi joacă optim (prin joc optim se înțelege că *Gicu* va încerca să maximizeze diferența de scor, în timp ce *Nicu* va încerca să o minimizeze), și că *Gicu* va începe jocul, determinați poziția inițială a jetonului, astfel încât diferența de scor între *Gicu* și *Nicu* să fie maximă!

Prima linie a fișierului *joc.in* conține două numere întregi  $N$  și  $M$ , separate prin câte un spațiu, care reprezintă numărul de linii și coloane ale matricii. Următoarele  $N$  linii conțin câte  $M$  numere întregi, separate prin câte un spațiu, care descriu matricea.

Fișierul *joc.out* va conține trei numere întregi separate prin câte un spațiu: diferența maximă de scor între *Gicu* și *Nicu* și linia și coloana unde se va plasa jetonul la începutul jocului.

*Exemplu:*

joc.in	joc.out
1 6 2 1 3 4 0 5	3 1 -6

5. (\*\*\*\*\*+) *Gigel* este o persoană cu o imagine foarte bogată, mai ales când doarme! Într-o noapte a visat că are de înălținit o sarcină foarte bizată: trebuie să aleagă o secvență (adică un subșir de elemente care apar pe poziții consecutive în sirul inițial) din  $N \leq 30.000$  elemente pentru care se cunosc costul și timpul. Secvența aleasă trebuie să fie de lungime minimă  $L$  și maximă  $U$ , iar suma costurilor elementelor secvenței, împărțită la suma timpurilor elementelor secvenței, să fie maximă. În scurt timp, visul lui *Gigel* s-a transformat într-un coșmar deoarece nu poate să rezolve sarcina! Scrieți un program care să-l ajute!

Pe prima linie în fișierul de intrare *secv3.in*, se află numere  $N$ ,  $L$  și  $U$  separate prin câte un spațiu. Pe cea de a doua linie se vor găsi  $N$  numere naturale reprezentând costurile elementelor secvenței, iar pe cea de a treia linie se vor găsi  $N$  numere naturale reprezentând timpurile elementelor secvenței. Costul și timpul unui element este un număr natural din intervalul  $[1, 1.000]$ .

Pe prima linie din fișierul de ieșire *secv3.out* se va găsi un număr real cu precizie de două zecimale, reprezentând valoarea maximă a sumei costurilor elementelor din secvență împărțită la suma timpurilor elementelor din secvență.

*Exemplu:*

*secv3.in*

5 1 2  
1 1 3 2 5  
4 2 5 3 6

*secv3.out*

0.83

6. (\*\*\*) Lui *Gigel* îi place să se joace, aşa că el a primit un joc cu cutii pe care erau scrise numere întregi strict pozitive. Având o imagine bogată, el s-a gândit într-o zi să construiască un triunghi de cutii astfel: o cutie este așezată peste două cutii, care sunt așezate pe trei cutii... care sunt așezate pe  $N \leq 18$  cutii (deci fiecare cutie se sprijină pe alte două). Dar *Gigel* nu se oprește aici, ci vrea ca numărul din fiecare cutie din triunghiul său (mai puțin ultima linie) să fie egal cu suma numerelor din cele două cutii de dedesubt.

Scrieți un program care determină dacă *Gigel* poate construi un triunghi de latură  $N$ , în care suma numerelor de pe toate cutiile să fie  $S \leq 1.000.000$ , știind că poate folosi oricâte cutii cu orice număr în ele.

Pe prima linie, în fișierul *triunghi.in*, se vor afla numerele  $N$  și  $S$  separate prin câte un spațiu.

Pe primele  $N$  linii din fișierul *triunghi.out* se vor afla numerele de pe cutiile din triunghi: pe linia  $i$  vor fi scrise câte  $i$  numere întregi strict pozitive care descriu linia  $i$  din triunghi. Dacă *Gigel* nu poate construi un astfel de triunghi, în fișier se va afișa în schimb mesajul *imposibil*.

*Exemplu:*

*triunghi.in*

3 34

*triunghi.out*

13  
6 7  
1 5 2

7. (\*\*\*\*\*) *Gigel*, primar în orașul său, s-a gândit să renoveze strada principală, strada de dimensiuni  $M \times N$  ( $N \leq 150$ ,  $M \leq 15$ ) compusă din bucăți de dimensiuni  $1 \times 1$ . Majoritatea bucăților sunt stricate, dar mai există  $K$  bucăți care sunt considerate bune.

Dorind să plătească cât mai puțini bani, *Gigel* a luat de la un negustor blocuri de dimensiuni  $2 \times 2$  la prețul unui bloc de dimensiuni  $1 \times 1$ . Pentru a pava strada trebuie să amplaseze cât mai multe din aceste blocuri pe bucăți stricate, fără să paveze vreo bucăță bună deoarece ar apărea denivelări, și fără să se suprapună blocurile  $2 \times 2$ . El și-a dat seama că mai bine ar fi cumpărat blocuri  $1 \times 1$ , pentru că ar fi acoperit toată strada fără bătăi de cap, dar acum nu mai are de ales și are nevoie de ajutorul tău! Determinați numărul maxim de blocuri  $2 \times 2$  pe care le poate pune primarul pentru a repara strada.

Pe prima linie din fișierul *pavare.in* se vor afla trei numere întregi separate prin câte un spațiu:  $N$ ,  $M$  și  $K$ . Pe următoarele  $K$  linii se vor afla perechi de numere întregi reprezentând linia și coloana pe care se află o bucăță bună.

Pe prima linie, în fișierul *pavare.out*, se va afișa un număr natural reprezentând numărul maxim de blocuri  $2 \times 2$  care pot fi amplasate pe stradă.

*Exemplu:*

*pavare.in*

4 6 3  
1 1  
2 6  
3 3

*pavare.out*

4

8. (\*\*\*\*\*) *Zăhărel* s-a dus în vizită la bunicii lui de la țară împreună cu *Eugenia*. Ferma bunicilor este de formă circulară, iar acolo cresc  $N \leq 10.000$  găini. Ferma a fost împărțită în  $N$  sectoare, numerotate de la 1 la  $N$ , astfel încât oricare două sectoare având numere consecutive sunt adiacente (se află unul lângă altul). În plus, primul și ultimul sector sunt adiacente. În fiecare sector se află câte o găină, iar aceasta depune ouă în fiecare zi. Bunicii știu pentru fiecare găină care este productivitatea ei, adică un număr care reprezintă diferența dintre consumul găinii și cât produce. *Zăhărel* și cu *Eugenia* strâng de  $K \leq 1.000$  ori pe zi ouă de la găinii astfel: la fiecare strângere cei doi aleg o secvență (adică un sir de sectoare adiacente) formată din cel puțin un sector, care conține numai găini de la care nu s-au luat ouă în acea zi. Cunoșcând productivitatea fiecărei găini cât și numărul de strângeri dintr-o zi, ajutați-i pe *Zăhărel* și *Eugenia* să strângă ouă astfel încât suma productivităților găinilor de la care a strâns ouă să fie maximă.

Pe prima linie din *ferma.in* se găsesc numerele naturale  $N$  și  $K$ . Pe următoarea linie se vor găsi  $N$  numere întregi reprezentând productivitatea fiecărei găini. Productivitatea fiecărei găini este un număr întreg din intervalul [-100.000, 100.000].

Pe prima linie din *ferma.out* se va afișa suma maximă posibilă a productivităților găinilor de la care se strâng ouă.

*Exemplu:*

*ferma.in*

8 2  
2 -6 4 3 -7 -9 10 -1

*ferma.out*

18

9. (\*\*\*\*\*) *Bronzărel* a ieșit din spital și este sănătos acum. Imediat după ce a ieșit s-a întâlnit din nou cu bunul său prieten *Zăhărel* și acestia s-au pus pe rezolvat probleme! Una din problemele pe care au încercat să o rezolve s-a dovedit prea dificilă pentru ei, de aceea vor avea nevoie de ajutorul tău. Fie  $A$  o matrice de numere naturale cu  $N \leq 150$  linii și  $M \leq 150$  coloane. Vom defini o matrice  $B$  de mărime  $P \times Q$  ca fiind o submatrice a matricei  $A$  dacă există numerele  $(x, y)$  astfel încât  $B_{i,j} = A_{i+x, j+y}$ , pentru  $1 \leq i \leq P$  și  $1 \leq j \leq Q$ . De asemenea, vom defini balansul unei matrice ca fiind raportul dintre suma tuturor elementelor din matrice și numărul acestora.

Problema la care s-au chinuit *Zăhărel* și *Bronzărel* cere să se determine o submatrice de balans maxim a matricei  $A$ , care are cel puțin  $R$  rânduri și  $C$  coloane. Fiindcă lucrurile nu sunt niciodată așa de "simple", matricea  $A$  nu este o matrice oarecare, ci una chiar foarte specială, și anume rândurile și coloanele matricei pot fi permute circulare.

Determinați submatricea de balans maxim dintr-o matrice dată, ținând cont că rândurile și coloanele pot fi permute circular înainte, pentru a obține un rezultat cât mai favorabil.

Prima linie din fișierul *balans.in* va conține numerele *N,M,R,C* reprezentând dimensiunea matricei *A* și limitele inferioare ale dimensiunilor submatricei cerute. Următoarele *N* linii vor conține câte *M* numere naturale. Fișierul *balans.out* va conține pe o singură linie o valoare reprezentând balansul maxim posibil al unei submatrice. Rezultatul va fi afișat cu 3 zecimale exacte.

*Exemplu:*

<i>balans.in</i>	<i>balans.out</i>
3 4 2 1 15 5 15 8 1 2 1 3 4 8 8 4	11.500

10. (\*\*\*\*\*\*) *Zăhărel* se ocupă de securitatea subiectelor de la finala *preONI* și a scris textul problemelor sub forma unui sir *S<sub>0</sub>* de numere naturale ordonate crescător. Fieind un pic paranoic, el s-a gândit să aplique o criptare pe *S<sub>0</sub>*, folosind un algoritm inventat de el, algoritmul de criptare *PedeFé*. Din păcate, calculatorul *preONI* a fost virusat și datele criptate s-au pierdut. *Zăhărel* a putut recupera trei siruri de date din calculator pe care le vom nota *S<sub>1</sub>, S<sub>2</sub>, S<sub>3</sub>*. După câteva investigații și-a dat seama că *S<sub>3</sub>* este un subșir al lui *S<sub>0</sub>*, și împreună *S<sub>0</sub>* se găsește ca subșir atât în *S<sub>1</sub>* cât și în *S<sub>2</sub>*.

Evident, cele trei siruri nu sunt deajuns pentru a-l găsi în mod unic pe *S<sub>0</sub>*. Ca să nu-i irosească zilele să recuperze datele, *Zăhărel* vrea să știe câte posibilități există de a-l găsi pe *S<sub>0</sub>* folosind informațiile disponibile. Reamintim că *S<sub>0</sub>* trebuie să fie un subșir al lui *S<sub>1</sub>* și *S<sub>2</sub>*, să-l conțină pe *S<sub>3</sub>* ca subșir și să fie crescător.

Prima linie a fișierului *pedefe.in* va conține pe prima linie numerele naturale *N, M≤500* și *P≤100*. Următoarele trei linii vor conține *N, M*, respectiv *P* numere naturale, reprezentând sirurile *S<sub>1</sub>, S<sub>2</sub>, S<sub>3</sub>*, în această ordine. Sirurile vor conține numere naturale din intervalul [1, 500].

În fișierul *pedefe.out* se va afișa numărul de posibilități căutat, modulo 666013.

*Exemplu:*

<i>pedefe.in</i>	<i>pedefe.out</i>
8 9 2 14 1 2 2 15 24 3 4 17 18 1 2 2 3 24 4 19 1 24	6

11. (\*\*\*\*\*) Fie sirurile *a[1], a[2], a[3], ..., a[n]* și *b[1], b[2], b[3], ..., b[m]*, *m>n*. Să se maximizeze valoarea expresiei *E=a[1]x[1]+a[2]x[2]+...+a[n]x[n]*, unde *x[i]* sunt elemente ale sirului *b*.

Datele de intrare se vor citi din fișierul *valmax.in* în formatul următor: pe prima linie numerele *n* și *m*, iar pe următoarele două linii, elementele celor două siruri. Rezultatul va fi afișat pe ecran.

*Exemplu:*

Pentru *n=6, m=8* și sirurile *a=(3, 7, -10, 5, -1, 2)* respectiv *b=(10, 5, 20, -20, -2, 7, 9, -10)*, valoarea maximă a lui *E* este 441.

12. (\*\*\*\*\*) Se dă un sir de *N<15001* numere întregi (între -30000 și 30000). O secvență a acestui sir este alcătuiră din *M* elemente (numere) consecutive (*0<M≤N*). Să se scrie secvența a cărei sumă a elementelor este minimă, în modul. Din fișierul *dif.in* se citește de pe prima linie *N*, iar pe următoarele *N* linii se află elementele sirului mare (în ordine). În fișierul *dif.out* se va scrie un singur număr reprezentând suma (în modul) minimă.

*Exemplu:*

<i>dif.in</i>	<i>dif.out</i>
3 2 -3 4	1

13. (\*\*\*\*) Se consideră *n* pitici, care stau aliniați în rând, fiecare cu față spre spatele celuilalt. Piticii au pe cap căciulițe roșii și negre. Piticii cu căciulițe roșii spun întotdeauna adevarul, în timp ce piticii cu căciulițe negre mint întotdeauna. Fiecare pitic este întrebat câte căciulițe roșii vede în față sa. În funcție de răspunsul piticilor, trebuie să stabiliți ce culoare are căciulița lor. Din fișierul *pitici.in* se citește răspunsul piticilor. Formatul fișierului este: pe prima linie se află numărul de pitici *n* (*n<20000*). Pe următoarele linii se află perechi de câte două numere care reprezintă: număr pitic – răspunsul la întrebare (separate prin spațiu).

Răspunsul se va scrie în fișierul *pitici.out* pe o singură linie, fiind format dintr-o secvență de *n* caractere *R* sau *N*, *R* reprezentând culoarea roșie și *N*, culoarea neagră.

*Exemplu:*

<i>pitici.in</i>	<i>pitici.out</i>
5 3 2 4 1 2 2 5 3 1 0	RNNRN

14. (\*\*\*\*) Gigel are *N* cutii cu bile roșii, verzi și albastre. Într-o zi, el se hotărăște să strângă toate bilele în trei cutii: cele roșii într-o, cele verzi în alta și cele albastre în alta, diferită de primele două. În acest scop, el dorește să mute cât mai puține bile din cutiile în care sunt inițial, în cele în care vor ajunge. Ajutați-l! Fișierul de intrare *bile.in* are următoarea structură:

- N* - nr. de cutii;  $3 \leq N \leq 1000$ ;
- R1 V1 A1* - câte bile roșii, verzi și albastre are în prima cutie;
- R2 V2 A2* - ... și tot așa pentru celelalte cutii;
- ...

*RN VN AN*

Fișierul de ieșire *bile.out* are următoarea structură:

*NR* - numărul de bile mutate;

*CR CV CA* - numerele cutiilor în care ajung bilele roșii, verzi, respectiv albastre.

*Exemplu:*

bile.in

3  
3 4 1  
1 2 2  
0 5 0

bile.out

4  
1 3 2

15. (\*\*\*) Se consideră  $N$  puncte în plan ( $N \leq 10000$ ), cu coordonate întregi (între -30000 și 30000). Dintre acestea unele sunt dominante, iar altele nu. Un punct se consideră dominant dacă la dreapta lui (cu coordonata  $X$  cel puțin egală cu  $z$  lui) nu există nici un punct mai înalt decât el (cu coordonata  $Y$  mai mare, strict mai mare). Cerința voastră este să stabiliți căte din cele  $N$  puncte sunt dominante.

Fisierul dom.in conține pe prima linie numărul de puncte. Pe următoarele  $N$  linii se află coordonatele  $X$  și respectiv  $Y$  ale punctelor, separate printr-un spațiu.

În fisierul dom.out trebuie scrise căte dintre acestea sunt dominante.

*Exemplu:*

dom.in

3  
2 0  
0 1  
0 2

dom.out

2

16. (\*\*) Zăhărel are  $N \leq 5000$  vaci și pentru fiecare se știe intervalul de timp în care acestea produc lapte. Determinați care este cel mai lung interval de timp în care cel puțin o vacă produce lapte, și cel mai lung interval de timp în care nici o vacă nu produce lapte. În fisierul de intrare lapte.in va fi pe prima linie numărul  $N$ . Pe următoarele  $N$  linii se găsesc perechi de numere întregi  $\leq 1.000.000$ , reprezentând intervalele. În fisierul de ieșire lapte.out se vor scrie două numere reprezentând lungimea celui mai lung interval de timp în care cel puțin o vacă produce lapte, și lungimea celui mai lung interval de timp în care nici o vacă nu produce lapte.

*Exemplu:*

lapte.in

3  
300 1000  
700 1200  
1500 2100

lapte.out

900 300

17. (\*\*) Pe un panou dreptunghiular sunt dispuse  $N \times M$  beculete. Asociat panoului cu becuri, există un panou de comutatoare pentru aprinderea/stingerea becurilor. Totuși, sistemul de funcționare al comutatoarelor este puțin bizar. Astfel, prin acționarea unui comutator se va schimba nu numai starea becului asociat acestuia, ci și starea becurilor vecini pe linie și coloană.

Pe panou există becuri aprinse sau stinse. Pornind de la o configurație dată, se cere să se determine numărul minim de acționări de comutatoare care va realiza stingerea tuturor becurilor.

Pe prima linie a fisierului beculete.in se află  $N$  și  $M$  ( $1 < N, M \leq 16$ ). Pe următoarele  $N$  linii ale fisierului se află, nesperate prin spații, căte  $M$  valori din mulțimea {0,1} reprezentând starea becurilor: 1-aprins, 0-stins.

În fisierul beculete.out se va afișa numărul minim de acționări de comutatoare. Dacă nu există soluție, în fisier se va afișa mesajul "NU EXISTA".

*Exemplu:*

beculete.in

4 4  
0111  
1100  
0101  
1110

beculete.out

4

18. (\*\*) Gigel este pasionat de informatică și mai ales de cifrele 0 și 1; așa de mult, încât a atribuit fiecărui din cei  $2^N$  prieteni ai lui căte o etichetă, sub forma unui sir de biți de lungime  $N \leq 20$ . Toate etichetele sunt distințe între ele.

Gigel s-a gândit într-o zi că vrea să construiască o etichetă pentru el însuși, de lungime cât mai mică, care să conțină o singură dată, ca o subsecvență, fiecare din cele  $2^N$  etichete ale prietenilor lui. Scrieți un program care determină eticheta lui Gigel, de lungime minimă. Pe prima linie a fisierului de intrare biti.in se va găsi numărul  $N$ . Pe prima linie a fisierului de ieșire biti.out se va găsi lungimea sirului. Pe a doua linie se va afișa un sir de biți 0 sau 1 care vor reprezenta eticheta găsită.

*Exemplu:*

biti.in

3

biti.out

10  
0001011100

19. (\*\*\*\*\*) Bronzărel a crescut mare și în curând va merge la facultate. Totuși, admiserea la facultate nu este așa de simplă, având probe dificile precum matematică. Ca să se pregătească, Bronzărel lucrează zilnic la matematică, efectuând diverse calcule. Zăhărel vrea să-i arate lui Bronzărel că poate rezolva orice problemă de matematică cu ajutorul calculatorului și a îndemnării lui de programator și i-a cerut acestuia să-i dea să rezolve cea mai grea problemă pe care o știe! Bronzărel a scris imediat pe o foaie de hârtie următoarea sumă:

$$S(A, B) = A^1 + A^2 + A^3 + \dots + A^B$$

și i-a spus că trebuie doar să calculeze valoarea ei. Fiindcă rezultatul poate fi un număr foarte mare, Bronzărel se mulțumește dacă Zăhărel determină doar ultimele  $C$  cifre ale sumei.

Imaginați-vă că sunteți în locul lui Zăhărel și scrieți programul care îi va arăta lui Bronzărel că problemele dificile de matematică pot fi rezolvate cu ajutorul calculatorului!

Prima linie a fisierului calcul.in va conține numărul natural  $A < 10^{100.000}$ , în baza 10. A doua linie va conține numărul natural  $B < 16^{50.000}$ , care va fi dat în baza 16, iar a treia linie va conține numărul natural  $C \leq 9$ . Prima linie a fisierului calcul.out va conține ultimele  $C$  cifre ale sumei menționate mai sus.

*Exemplu:*

calcul.in

2  
7  
2

calcul.out

54

20. (\*\*\*\*\*) De-a lungul unei şosele, care deagă vârful cu baza unui deal sunt  $n$  copaci. Administrația locală a decis să-i iaie. Pentru a nu irosi lemn, copacii vor fi prelucrați cu mașini speciale. Copacii pot fi transportați doar de sus în jos. În partea de jos a şoselei este o mașină de prelucrat lemn. Două alte mașini de prelucrat lemn pot fi construite de-a lungul şoselei. Trebuie să decideți unde trebuie să fie construite aceste mașini de prelucrat lemn astfel încât costul de transport să fie minim. Costul transportului este de un cent pentru un kilogram de lemn transportat un metru. Scrieți un program, care calculează costul minim de transport.

Prima linie a fișierului de intrare *copaci.in* conține un număr întreg  $n$  – numărul de copaci ( $2 \leq n \leq 20000$ ). Copacii sunt numerotati 1, 2, ...,  $n$ , pornind din vârful dealului și mergând spre vale. Fiecare dintre următoarele  $n$  linii conține două numere întregi pozitive separate prin câte un spațiu. Linia  $i+1$  conține:  $m_i$  – greutatea (în kilograme) a copacului  $i$  și  $d_i$  – distanța (în metri) între copacii numerotati cu  $i$  și  $i+1$ , ultimul dintre aceste numere  $d_n$  reprezintă distanța dintre copacul  $n$  și vale (de unde începe șoseaua). Este garantat că costul total de transport al copacilor până la mașina de prelucrat lemn din vale este mai mic decât 2000000000.

Prima și singura linie din fișierul *copaci.out* va conține un număr întreg care reprezintă costul minim de transport.

*Exemplu:*

*copaci.in*

```
9
1 2
2 1
3 3
1 1
3 2
1 6
2 1
1 2
1 1
```

26

*copaci.out*

(CEOI 2004)

21. (\*\*\*\*) Considerăm axa *OX* cu originea în punctul de coordonată 0. Pe această axă nu pot fi reprezentate decât puncte având coordonate naturale mai mici decât 20000. Originea reprezintă un punct de pe axă.

Cunoscându-se distanțele dintre *oricare două puncte* reprezentate pe axă, se cere determinarea coordonatelor acestora. Lista distanțelor cuprinde cel mult 62000 de valori. În fișierul text *puncte.in* sunt scrise, pe prima linie, distanțele dintre punctele reprezentate pe axă, separate prin câte un spațiu. Fișierul de ieșire *puncte.out* va conține o singură linie pe care se vor scrie coordonatele punctelor reprezentate pe axă, în ordinea crescătoare a valorilor. În cadrul liniei, numerele vor fi separate prin câte un spațiu.

*Exemplu:*

*puncte.in*

```
10
2 4 6 2 5 7 9 9 11 3
```

*puncte.out*

0 2 6 9 11

## Indicații și răspunsuri

### Sectiunea 1.1.1

1. a), c)	5. a), d)	9. d)	13. a)	17. a)
2. b), d)	6. c)	10. c)	14. a)	18. c), d)
3. d)	7. d)	11. a), c)	15. c)	19. a), d)
4. a, c)	8. b), d)	12. b), c)	16. b), d)	20. b), c)

### Sectiunea 1.1.3

```

1. procedure move(var p:point);
2. var q:point;
3. begin
4.   q := p;
5.   while q^.leg<>nil do q:=q^.leg;
6.   q^.leg := p;
7.   p := p^.leg^.leg;
8.   q^.leg^.leg^.leg := nil
9. end;
10.
11. void move()
12. {
13.   point *q;
14.   q=p;
15.   while (q->leg!=NULL) q=q->leg;
16.   q->leg=p;
17.   p=p->leg->leg;
18.   q->leg->leg->leg=NULL;
19. }
20.
21. function nr(p:point):byte;
22. var min:point; i:byte;
23. begin
24.   min := p;
25.   i := 0;
26.   while p <> nil do begin
27.     if p^.inf<min^.inf then begin
28.       min := p;
29.       nr := i;
30.     end;
31.     p := p^.leg;
32.     inc(i);
33.   end;
34. end;
35.
36. int nr(point *p) {
37.   point *min,int i,rez=0;
38.   min=p;
39.   i=0;
40.   while (p!=NULL) {
41.     if (p->inf<min->inf) {
42.       min=p;
43.       rez=i
44.     }
45.     p=p->leg;
46.     i++;
47.   }
48.   return rez;
49. }
50.
51. procedure del(p:point);
52. var t, q:point; x:byte;
53. begin
54.   t:=p; p:=p^.leg; x:=t^.inf;
55.   while p^.leg<>nil do
56.     if p^.inf=(x+p^.leg^.inf)/2
57.     then begin
58.       q := p; t^.leg := p^.leg;
59.       p := p^.leg; x := q^.inf;
60.       dispose(q);
61.     end else begin
62.       t:=p; p:=p^.leg; x:=t^.inf;
63.     end;
64.   end;
65. }
```

```

6.
1 procedure del(p,u:pointd);
2 var t,q:pointd;
3 begin
4 while (p^.ld<u^.ls) and
5 (p^.ld>u)
6 do begin
7   p:=p^.ld;
8   u:=u^.ls;
9 end;
10 if p^.ld=u^.ls then begin
11   q:=p^.ld;
12   p^.ld:=u;
13   u^.ls:=p;
14   dispose(q);
15 end else begin
16   q:=p; t:=u;
17   p^.ls^.ld := u^.ls;
18   u^.ld^.ls := p^.ls;
19   dispose(q); dispose(t)
20 end
21 end;

```

```

3.
1 procedure X(p,q:point;var t:point);
2 var s:point;
3 begin
4 new(s);
5 if p^.inf < q^.inf then t:=p
6 else t:=q;
7 while (q<nil) and (p<nil) do
8 if p^.inf < q^.inf then begin
9   s^.leg:=p; s:=p; p:=p^.leg;
10 end else begin
11   s^.leg:=q; s:=q; q:=q^.leg;
12 end;
13 if p<nil then s^.leg:=p
14 else s^.leg:=q;
15 end;

```

```

10.
1 procedure D(var p:point);
2 var t,q:point;
3 begin
4   t:=p;
5   repeat
6     p:=p^.leg
7     until p^.leg^.leg=t;
8     q:=p^.leg;
9     p^.leg:=t^.leg;
10    p:=t^.leg;
11    dispose(t);
12    dispose(q)
13 end;

```

```

void del(pointd *p, pointd *u)
{
  pointd *t,*q;
  while ((p->ld!=u->ls)&&(p->ld!=u))
  {p=p->ld;
   u=u->ls;
  }
  if (p->ld==u->ls)
  {q=p->ld;
   p->ld=u;
   u->ls=p;
   delete(q);
  } else {
   q=p;
   t=u;
   p->ls->ld=u->ld;
   u->ld->ls=p->ls;
   delete(q);
   delete(t);
  }
}

```

```

void X(point *p, point *q, point *&t) {
  point *s;
  s=new point;
  if (p->inf < q->inf) t=p;
  else t=q;
  while (q!=NULL&&p!=NULL)
  {if (p->inf < q->inf) {
    s->leg=p; s=p; p=p->leg;
  } else {
    s->leg=q; s=q; q=q->leg;
  }
  if (p!=NULL) s->leg=p;
  else s->leg=q;
}

```

```

void D(point *&p)
{
  point *t,*q;
  t=p;
  do p=p->leg; while (p->leg->leg!=t);
  q=p->leg;
  p->leg=t->leg;
  p=t->leg;
  delete t;
  delete q;
}

```

```

13.
1 procedure S(p:point;
2   var a,b:point; x:integer);
3 var s1,s2:point;
4 begin
5   new(s1);new(s2);a:=s1; b:=s2;
6   while p<nil do begin
7     if p^.inf<x then begin
8       s1^.leg:=p; s1:=p;
9     end
10    else begin
11      s2^.leg:=p; s2:=p;
12    end;
13    p:=p^.leg;
14  end;
15  s1^.leg=nil; s2^.leg=nil;
16  a:=a^.leg; b:=b^.leg;
17 end;

```

```

22.
1 procedure creare;
2 var p:point; i,j:byte;
3 begin
4   read(n);
5   for i:=1 to n do a[i]:=nil;
6   for i:=1 to n do begin
7     read(x);
8     for j:=1 to n do begin
9       new(p); p^.inf:=j*x;
10      p^.leg:=a[i]; a[i]:=p;
11    end;
12  end;
13 end;

```

```

26.
1 procedure Make(p:point);
2 var u,t,temp:point;
3 use:array[0..9] of boolean;
4 begin
5   fillchar(use,10,false);
6   u:=p; temp:=p; p=p^.leg;
7   use[temp^.inf]:=true;
8   while p<nil do
9     if use[p^.inf] then begin
10       u^.leg:=p^.leg; t:=p;
11       p:=p^.leg; dispose(t);
12     end
13     else begin
14       use[p^.inf]:=true;
15       u:=p;
16       p:=p^.leg;
17     end;
18     u^.leg:=temp;
19 end;

```

```

void S(point *p, point *&a,
      point *&b, int x) {
  point *s1,*s2;
  s1=new point; s2=new point;
  a=s1; b=s2;
  while (p!=NULL)
  {
    if (p->inf<=x) {
      s1->leg=p; s1=p;
    } else {
      s2->leg=p; s2=p;
    }
    p=p->leg;
  }
  s1->leg=s2->leg=NULL;
  a=a->leg; b=b->leg;
}

```

```

void creare()
{
  point *p;int i,j,x;
  scanf("%d",&n);
  for(i=1;i<=n;i++) a[i]=NULL;
  for(i=1;i<=n;i++) {
    scanf("%d",&x);
    for (j=1;j<=n;j++) {
      p=new point; p->inf=j*x;
      p->leg=a[i]; a[i]=p;
    }
  }
}

```

```

void make(point *p)
{
  point *u,*t,*temp;
  int use[10];
  memset(use,0,sizeof(use));
  u=p; temp=p; p=p->leg;
  use[temp->inf]=1;
  while (p!=NULL)
  {if (use[p->inf]) {
    u->leg=p->leg;
    t=p; p=p->leg;
    delete t;
  } else {
    use[p->inf]=1;
    u=p;
    p=p->leg;
  }
  u->leg=temp;
}
}

```

## Sectiunea 1.2.2

```

27.
1 function ok(p:pointd;x:real);
2         pointd;
3 var t:pointd;
4 begin
5 if x<p^.inf then ok:=nil
6 else begin
7     t:=p;
8     while (p<>nil)and(x>p^.inf) do
9         begin t:=p; p:=p^.ld; end;
10    ok:=t
11 end
12 end;
13
14 function creare:pointd;
15 var q,p:pointd;
16     x,y,z:real;
17 begin
18 new(prim); read(x);
19 prim^.inf:=x; prim^.ld:=nil;
20 prim^.ls:=nil; read(z); y:=x+z;
21 while z<>(y+x)/2 do begin
22     q:=ok(prim,z);
23     if q<>nil then begin
24         new(p); p^.inf:=z;
25         p^.ld:=q^.ld; p^.ls:=q;
26         q^.ld:=p; q^.ld:=p;
27     end else begin
28         new(p); p^.inf:=z;
29         p^.ld:=q; p^.ls:=nil;
30         q^.ls:=p; end;
31         x:=y; y:=z; read(z);
32     end;
33 creare:=prim;
34 end;

```

```

29.
1 .....
2 procedure S(prim,p:point;x:byte);
3 begin
4 if (p<>prim) then begin
5     S(prim,p^.leg,x);
6     write(p^.inf,' ');
7 end
8 else
9 if x=0 then begin
10     S(prim,p^.leg,1);
11     write(p^.inf,' ');
12 end;
13 end;
14
15 begin
16 .....
17 S(prim,prim,0); {spal}
18 end.

```

```

pointd* ok(pointd *p,double x)
{
    pointd *t;
    if (x<p->inf) return NULL;
    t=p;
    while (p!=NULL&&x>p->inf)
    {
        t=p; p=p->ld;
    }
    return t;
}

pointd* creare()
{
    pointd *p,*q;double x,y,z;
    prim=new pointd;scanf("%lf",&x);
    prim->inf=x;prim->ls=prim->ld=NULL;
    scanf("%lf",&z);y=x+z;
    while (z!= (y+x)/2) {
        q=ok(prim,z);
        if (q!=NULL)
        {
            p=new pointd;p->inf=z;
            p->ld=q->ld;p->ls=q;
            q->ld->ls=p;q->ld=p;
        } else {
            p=new pointd;p->inf=z;
            p->ld=q;p->ls=NULL;
            q->ls=p;
        }
        x=y;y=z; scanf("%lf",&z);
    }
    return prim;
}

```

```

..... void S(point *prim,point *p,int x)
{
    if (p!=prim)
    {
        S(prim,p->leg,x);
        printf("%d ", p->inf);
    } else
    if (!x)
        S(prim,p->leg,1);
        printf("%d ", p->inf);
    }
}

int main() {
    .....
    S(prim,prim,0);
}

```

1. b),c)	5. b),c)	9. d)	13. b)	17. a)
2. a), d)	6. d)	10. c)	14. c)	18. b)
3. c)	7. c)	11. b),c)	15. a),c)	19. b)
4. a), c)	8. a),b)	12. a)	16. a),d)	20. c)

## Sectiunea 1.2.4

```

1. 1 procedure tree(var p:varf);
2 var x:byte;
3 begin
4     read(x);
5     if x = 0 then p := nil
6     else begin
7         new(p);
8         p^.inf := x;
9         tree(p^.fs);
10        tree(p^.fd);
11    end;
12 end;

```

```

void tree(varf *&p)
{
    int x;
    scanf("%d",&x);
    if (!x) {
        p=NULL; return;
    }
    p=new varf;
    p->inf=x;
    tree(p->fs);
    tree(p->fd);
}

```

```

2. 1 procedure InO(p:varf;nv:byte);
2 begin
3     if p<>nil then begin
4         InO(p^.fs,nv + 1);
5         if nv mod 2 = 0 then
6             write(p^.inf,' ');
7         InO(p^.fd,nv + 1);
8     end;
9 end;

```

```

void InO(varf *p,int nv)
{
    if (p!=NULL) {
        InO(p->fs,nv+1);
        if (nv%2==0)
            printf("%d ",p->inf);
        InO(p->fd,nv+1);
    }
}

```

```

4. 1 procedure PostO(p:varf);
2 begin
3     if p<>nil then begin
4         PostO(p^.fs);
5         PostO(p^.fd);
6         if (p^.fs<>nil) and
7             (p^.fd<>nil) then
8             write(p^.inf,' ');
9     end;
10 end;

```

```

void PostO(varf *p)
{
    if (p!=NULL)
    {
        PostO(p->fs);PostO(p->fd);
        if (p->fs!=NULL&&p->fd!=NULL)
            printf("%d ",p->inf);
    }
}

```

```

5. 1 procedure Search(p:varf;
2         var min,max:integer);

```

```

void Search(varf *p,
            int &min,int &max)

```

```

3 begin
4   if p>nil then begin
5     Search(p^.fs,min,max);
6     Search(p^.fd,min,max);
7     if p^.inf>max then
8       max:=p^.inf;
9     if p^.inf<min then
10      min:=p^.inf;
11    end;
12  end;

13 procedure Del(var p:varf);
14 begin
15   if p>nil then begin
16     if (p^.fs=nil)and(p^.fd=nil)
17     then p:=nil
18     else begin
19       Del(p^.fs);
20       Del(p^.fd);
21     end;
22   end;
23 end;

```

```

1 .....
2 function Plin(p:varf; nv:byte;
3   var m,ok:byte):integer;
4 var x,y:byte;
5 begin
6   if nv>m then m:=nv;
7   if (p^.fs<>nil)and(p^.fd<>nil)
8   then begin
9     x:=Plin(p^.fs,nv+1,m,ok);
10    y:=Plin(p^.fd,nv+1,m,ok);
11    if x+y+1<>1 shl(m-nv+1)-1
12    then ok:=0;
13    Plin:=x+y+1;
14  end
15  else
16    if (p^.fs=nil)and(p^.fd=nil)
17    then Plin:=1
18    else begin
19      ok:=0;
20      Plin:=2;
21    end;
22 end;
23 begin
24   .....
25   ok:=1;
26   m:=1;
27   Plin(root,1,m,ok);
28   write(ok);
29 end.

```

```

1 if !(p!=NULL) {
2   Search(p->fs,min,max);
3   Search(p->fd,min,max);
4   if !(p->inf>max)
5     max=p->inf;
6   if !(p->inf<min)
7     min=p->inf;
8 }
9
10 void Del(varf *&p){
11   if (p!=NULL){
12     if (p->fs==NULL&&p->fd==NULL)
13       p=NULL;
14     else {
15       Del(p->fs);
16       Del(p->fd);
17     }
18   }
19 }

```

```

1 .....
2 int Plin(varf *p,int nv,
3           int &m,int &ok)
4 {
5   int x,y;
6   if (nv>m) m=nv;
7   if (p->fs!=NULL&&p->fd!=NULL) {
8     x=Plin(p->fs,nv+1,m,ok);
9     y=Plin(p->fd,nv+1,m,ok);
10    if (x+y+1!=((1<<(m-nv+1))-1)) ok=0;
11    return x+y+1;
12  }
13  if (p->fs==NULL&&p->fd==NULL)
14    return 1;
15  ok=0;
16  return 2;
17 }

18 int main()
19 {
20   int m,ok;
21   .....
22   Plin(root,1,m,ok);
23   printf("%d\n",ok);
24   return 0;
25 }

```

```

12
1 .....
2 #procedure Equal(p,q:varf;
3   var ok:byte);
4 begin
5   if (p>nil)and(q>nil)
6   then
7     if p^.inf<>q^.inf then
8       ok:=0
9     else
10      begin
11        Equal(p^.fs,q^.fs,ok);
12        Equal(p^.fd,q^.fd,ok);
13      end
14  end;
15 begin
16   .....
17   ok:=1;
18   Equal(root1,root2,ok);
19   writeln(ok);
20 end.

```

```

void Equal(varf *p,varf *q,int &ok)
{
  if (p!=NULL&&q!=NULL)
    if (p->inf!=q->inf)
      ok=0;
    else
    {
      Equal(p->fs,q->fs,ok);
      Equal(p->fd,q->fd,ok);
    }
}

int main()
{
  .....
  ok=1;
  Equal(root1,root2,ok);
  printf("%d\n",ok);
  return 0;
}

```

```

17
1 procedure Post0(p:varf;
2   var x:integer);
3 begin
4   if (p <> nil) then
5     begin
6       Post0(p^.fs, k);
7       Post0(p^.fd, k);
8       dec(x);
9       if x=0 then
10         write(p^.inf,'')
11     end;
12  end;

```

```

void Post0(varf *p,int &x)
{
  if (p!=NULL) {
    Post0(p->fs,x);
    Post0(p->fd,x);
    x--;
    if (!x)
      printf("%d ",p->inf);
  }
}

```

```

18
1 function Ok(p:varf):byte;
2 begin
3   if p>nil then begin
4     if (p^.fs<>nil) and
5       (p^.inf<p^.fs^.inf)
6     then Ok:=0
7     else
8       if (p^.fd<>nil) and
9         (p^.inf>p^.fd^.inf)
10        then Ok:=0
11        else begin
12          Ok:=Ok(p^.fs)*Ok(p^.fd);
13        end;
14      end;
15    else Ok:=1;
16  end;

```

```

int Ok(varf *p)
{
  if (p!=NULL)
  {
    if (p->fs&&p->inf<p->fs->inf)
      return 0;
    else
      if (p->fd&&p->inf>p->fd->inf)
        return 0;
      else
      {
        return Ok(p->fs)*Ok(p->fd);
      }
    }
  return 1;
}

```

```

22.
1 .....  

2 procedure Del(var p:varf);  

3 begin  

4 if (p^.fs=nil)and (p^.fd=nil)  

5 then p:=nil  

6 else Del(p^.fs);  

7 end;  

8 begin  

9 ...  

10 Del(root^.fd)  

11 ...  

12 end.

```

```

.....  

void Del(varf *&p) {  

    if (p->fs==NULL&&p->fd==NULL)  

        p=NULL;  

    else Del(p->fs);  

}  

int main() {  

    ...  

    Del(root->fd);  

    ...
}

```

### Sectiunea 2.2.1

1. b)	6. b), c), d)	11. d)	16. a)	21. d)
2. d)	7. d)	12. c)	17. b)	22. a), d)
3. a)	8. b)	13. a)	18. b)	23. c)
4. c)	9. a)	14. b)	19. c)	24. b), d)
5. d)	10. c)	15. c)	20. c)	25. a-2,b-3, c-4,d-1

```

9 function Ok(G : mat; v : sir;  

10 k, n : byte):byte;  

11 var i:byte;  

12 use:array[1..25]of boolean;  

13 begin  

14 fillchar(use,sizeof(use),false);  

15 Ok := 1;  

16 use[v[1]] := True;  

17 for i := 1 to k - 1 do  

18 if (G[v[i]], v[i+1]) = 0 or  

19 use[v[i+1]] then Ok := 0  

20 else use[v[i+1]] := True;  

21 end;

```

```

int Ok(mat G,sir v,int k,int n)
{
    int i,use[26];
    memset(use,0,sizeof(use));
    use[v[1]]=1;
    for (i=1;i<k;i++)
        if (!G[v[i]][v[i+1]])|||use[v[i+1]])
            use[v[i+1]]=1;
    else use[v[i+1]]=1;
    return 1;
}

```

### 3.

```

1 type mat=array[1..25,1..25]of
2     0..1;
3     sir=array[1..25]of byte;
4     var G:mat; v:sir; m,n,k:byte;
5     .....
6     function Ok(G : mat; v : sir;  

7             k, n : byte):byte;  

8     var i:byte;  

9     begin  

10    Ok:=1;  

11    for i:=1 to k-1 do  

12    if (G[v[i]], v[i+1])<>1) then  

13        Ok:=0  

14    else begin  

15        G[v[i],v[i+1]] := 2;  

16        G[v[i+1],v[i]] := 2;  

17    end;  

18 end;

```

```

#include <stdio.h>
typedef char mat[26][26];
typedef int sir[26];
mat G;sir v;int m,n,k;
.....
int Ok(mat G,sir v,int k,int n)
{
    int i,..;
    for (i=1;i<k;i++)
        if (G[v[i]][v[i+1]]!=1)
            return 0;
        else
            G[v[i]][v[i+1]]=
            G[v[i+1]][v[i]]=2;
    return 1;
}

```

### 4.

```

1 type mat=array[1..25,1..25]of
2     0..1;
3     var G:mat; m,n:byte;
4     .....
5     function Ok(G:mat;  

6             n:byte):byte;  

7     var i,j:byte;  

8     begin  

9     Ok:=1;  

10    for i:=1 to n-1 do
11        for j:=i+1 to n do
12            if (G[i,j]<>1) then Ok:=0
13    end;

```

```

#include <stdio.h>
typedef char mat[26][26];
mat G;int m,n;
.....
int Ok(mat G,int n)
{
    int i,j;
    for (i=1;i<n;i++)
        for (j=i+1;j<=n;j++)
            if (G[i][j]!=1) return 0;
    return 1;
}

```

### 5.

```

1 type mat=array[0..25,0..25]of
2     0..1;
3     var G:mat; m,n:byte;
4     .....

```

```

#include <stdio.h>
typedef char mat[26][26];
mat G;int m,n;
.....

```

1.

```

1 type mat=array[1..25,1..25]of
2     0..1;
3     sir=array[1..25]of byte;
4     var G:mat; v:sir; m,n,k:byte;
5     .....
6     function Ok(G : mat; v : sir;  

7             k, n : byte):byte;  

8     var i:byte;  

9     begin  

10    Ok := 1;  

11    for i := 1 to k - 1 do
12    if G[v[i]], v[i + 1]) = 0
13    then Ok := 0;
14 end;

```

```

#include <stdio.h>
type mat=array[1..25,1..25]of
0..1;
sir=array[1..25]of byte;
var G:mat; v:sir; m,n,k:byte;
.....
function Ok(G : mat; v : sir;  

k, n : byte):byte
{
    int i;
    for (i=1;i<k;i++)
        if (G[v[i]][v[i+1]]!=1)
            return 0;
        else
            G[v[i]][v[i+1]]=
            G[v[i+1]][v[i]]=2;
    return 1;
}

```

2.

```

1 type mat=array[1..25,1..25]of
2     0..1;
3     sir=array[1..25]of byte;
4     var G:mat;
5     v:sir;
6     m,n,k:byte;
7     .....
8

```

```

#include <stdio.h>
#include <string.h>
type mat=array[1..25]of byte;
var G;
sir v;
int m,n,k;
.....

```

```

5. function Ok(G:mat; m:byte)
6.           byte;
7. var i,j:byte;
8. begin
9.   Ok:=1;
10.  for i:=1 to m do
11.    for j:=1 to n do
12.      inc(G[i,0], G[i,j]);
13.  for i:=2 to n do
14.    if G[i,0]<>G[1,0] then Ok:=0
15. end;

```

```

7.
1. type mat=array[0..25,0..25]of
2.     byte;
3. var G:mat; n:byte;
4.
5. function Ok(G:mat; m:byte)
6.           byte;
7. var i,j:byte;
8. begin
9.   Ok:=1;
10.  for i:=1 to m do
11.    for j:=i+1 to n do
12.      if (G[i][j]*G[j][i])<>1
13.        return 0;
14.  for i:=1 to n do
15.    if G[i,i]<>0 then Ok:=0
16. end;

```

```

9.
1. type
2. mat=array[0..25,0..25]of byte;
3. Plista:^lista;
4. lista = record nod : byte;
5.   urm : Plista; end;
6. sir=array[1..25]of Plista;
7. var G:mat; m,n:byte; L:sir;
8.
9. procedure Build(G:mat; n:byte;
10. var L:sir);
11. var i,j:byte;
12. p:Plista;
13. begin
14.   for i:=1 to n do L[i]:=nil;
15.   for i:=1 to n do
16.     for j:=i+1 to n do
17.       if G[i,j]=1 then begin
18.         new(p); p^.nod:=j;
19.         p^.urm:=L[i]; L[i]:=p;
20.         new(p); p^.nod:=i;
21.         p^.urm:=L[j]; L[j]:=p;
22.       end;
23.   end;

```

```

int Ok(mat G,int n)
{
  int i,j;
  for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
      G[i][0]+=G[i][j];
  for (i=2;i<=n;i++)
    if (G[i][0]!=G[1][0])
      return 0;
  return 1;
}

```

```

#include <stdio.h>
typedef char mat[26][26];
mat G; int m,n;
function Ok(G:mat; m:byte)
           byte;
var i,j:byte;
begin
  Ok:=1;
  for i:=1 to m do
    for j:=i+1 to n do
      if (G[i][j]*G[j][i])!=1
        return 0;
  for i:=1 to n do
    if (G[i][i])!=0 return 0;
  return 1;
}

```

```

#include <stdio.h>
typedef char mat[26][26];
struct lista {
  int nod;
  lista *urm;
};
typedef lista* sir[26];
mat G; int n,m; sir L;
procedure Build(G:mat; n:byte);
var i,j:byte;
p:Plista;
begin
  for i:=1 to n do L[i]:=nil;
  for i:=1 to n do
    for j:=i+1 to n do
      if (G[i][j]){
        p=new lista; p^.nod:=j;
        p^.urm:=L[i]; L[i]:=p;
        p=new lista; p^.nod:=i;
        p^.urm:=L[j]; L[j]:=p;
      }
}

```

```

11.
1. type mat=array[0..25,0..25]of
2.     byte;
3. sir=array[1..25]of boolean;
4. var G:mat; m,n:byte; use:sir;
5. .....
6. procedure solve;
7. var i,nr,x:byte;
8. begin
9.   fillchar(use,n,false);
10.  nr:=0; x:=0;
11.  for i:=1 to n do
12.    if not use[i] then begin
13.      Parcure_Df(i); inc(nr);
14.      if x>0 then writeln(x,' ',i);
15.      x:=i;
16.    end;
17.    writeln(nr-1)
18. end;

```

```

14.
1. type mat=array[0..25,0..25]of
2.     byte;
3. sir=array[1..25]of boolean;
4. var G:mat; i,m,n,ok:byte;
5. use:sir;
6. .....
7. procedure Df(x,tata:byte;
8.                 var ok:byte);
9. var i:byte;
10. begin
11.   use[x]:=true;
12.   for i:=1 to n do
13.     if G[x,i]=1 then
14.       if not use[i] then
15.         Df(i,x,ok);
16.       else if i<>tata then ok:=0;
17.   end;
18. end;
19. begin
20.   ..... ok:=1;
21.   for i:=1 to n do
22.     if not use[i] then
23.       Df(i,0,ok);
24.     writeln(ok)
25. end;

```

```

15.
1. type
2. mat=array[0..25,0..25]of byte;
3. sir=array[1..25]of integer;
4. var G:mat;
5. m,n:byte; deg:sir;
6. .....

```

```

#include <stdio.h>
#include <string.h>
typedef char mat[26][26];
typedef int sir[26];
mat G; int m,n; sir use;
.....
void solve(){
  int i,nr,x;
  memset(use,0,m);
  nr=x=0;
  for (i=1;i<=n;i++)
    if (!use[i]){
      Parcure_Df(i); nr++;
      if (x>0) printf("%d %d\n",x,i);
      x=i;
    }
  printf("%d\n",nr-1);
}

```

```

#include <stdio.h>
#include <string.h>
typedef char mat[26][26];
typedef int sir[26];
mat G; int i,m,n,ok; sir use;
.....
void DF(int x,int tata,int &ok)
{
  int i;
  use[x]=1;
  for (i=1;i<=n;i++)
    if (G[x][i]==1)
      if (!use[i]) DF(i,x,ok);
      else if (i!=tata) ok=0;
}
int main()
{
  ..... ok=1;
  for (i=1;i<=n;i++)
    if (!use[i]) DF(i,0,ok);
    printf("%d\n",ok);
  return 0;
}

```

```

#include <stdio.h>
#include <string.h>
typedef char mat[26][26];
typedef int sir[26];
mat G; int m,n; sir deg;
.....

```

```

73 procedure build(var G:mat;
    m:byte; deg:sir);
74 var x,i,j:byte;
75 begin
76 for i:=1 to n-1 do
77   for j:=i+1 to n do
78     if deg[i]>deg[j] then
79       begin
80         x:=deg[i];
81         deg[i]:=deg[j];
82         deg[j]:=x;
83       end;
84 for i:=1 to n-2 do begin
85   j:=i+1;
86   while j<=n do begin
87     if deg[j]>1 then begin
88       dec(deg[j]);
89       break;
90     end;
91     inc(j);
92   end;
93   G[j,i]:=1;
94   G[i,j]:=1;
95   writeln(i, ' ',j);
96 end;
97 G[n-1,n]:=1;
98 G[n,n-1]:=1;
99 writeln(n-1, ' ',n);
100 end;

```

```

16.
1 type
2 mat=array[0..25,0..25]of 0..1;
3 var G:mat; m,n:byte;
4 .....
5 Function Ok(G:mat;n:byte):byte;
6 var k,i,j,x:byte;
7 begin
8   Ok:=0;
9   for i:=1 to n-1 do
10     for j:=i+1 to n do begin
11       x:=0;
12       for k:=1 to n do
13         if (G[i,k]=1)and(G[j,k]=1)
14           then inc(x);
15       if x>1 then Ok:=1;
16     end;
17   end;

```

```

18.
1 type
2 mat=array[0..25,0..25]of 0..1;
3 var G:mat; m,n:byte;
4 use:array[1..100]of boolean;

```

```

void build(mat &G,int n,sir deg){
    int i,j,x;
    for (i=1;i<n;i++)
      for (j=i+1;j<=n;j++)
        if (deg[i]>deg[j])
        {
          x=deg[i];
          deg[i]=deg[j];
          deg[j]=x;
        }
    for (i=1;i<n-1;i++)
      j=1;
      while (j<=n) {
        if (deg[j]>1) {
          deg[j]--;
          break;
        }
        j++;
      }
    G[i][j]=G[j][i]=1;
    printf("%d %d\n",i,j);
  }
  G[n-1][n]=G[n][n-1]=1;
  printf("%d %d\n",n-1,n);
}

```

```

#include <stdio.h>
#include <string.h>
typedef char mat[26][26];
typedef int sir[26];
mat G;int i,m,n,ok;sir use;
.....
int Ok(mat G,int n){
  int k,i,j,x;
  for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++) {
      x=0;
      for (k=1;k<=n;k++)
        if (G[i][k]&&G[j][k]) x++;
      if (x>1) return 1;
    }
  return 0;
}

```

```

#include <stdio.h>
typedef char mat[26][26];
mat G; int m,n,use[101];

```

```

5 procedure df(x:byte);
6 var i:byte;
7 begin
8   use[x]:=True;
9   for i:=1 to n do
10     if (G[x,i]=1) then
11       if not use[i] then begin
12         G[i,x]:=0;
13         df(i);
14       end
15     else G[i,x]:=0;
16 end;

```

24.

```

1 type mat=array[0..25,0..25]of
2   0..1;
3 sir=array[1..25]of byte;
4 var G:mat; s:sir; m,n:byte;
5 use:array[1..100]of boolean;
6
7 procedure df(x,nv:byte);
8 var i:byte;
9 begin
10  use[x]:=True;
11  if odd(nv) then s[x]:=1
12  else s[x]:=2;
13  for i:=1 to n do
14    if (G[x,i]=1)and not use[i]
15    then df(i,nv+1)
16 end;

```

27.

```

1 type
2 mat=array[0..25,0..25]of 0..1;
3 sir=array[1..25]of byte;
4 var G:mat; s:sir; m,n:byte;
5
6 procedure solve;
7 var i,j,k,s:byte;
8 begin
9   for k := 1 to n do
10     for i := 1 to n do
11       for j := 1 to n do
12         if G[i,j] = 0 then
13           G[i,j]:= G[i,k] * G[k,j];
14   for j := 1 to n do begin
15     s:=0;
16     for i := 1 to n do
17       if i <> j then
18         s := s + G[i,j];
19     if s=n-1 then writeln(j,'');
20   end;
21 end;

```

```

void df(int x)
{
  int i;
  use[x]=1;
  for (i=1;i<=n;i++)
    if (G[x][i])
      if (!use[i]) {
        G[i][x]=0;
        df(i);
      } else G[i][x]=0;
}

```

```

#include <stdio.h>
typedef char mat[26][26];
typedef int sir[26];
mat G; int m,n,use[101];
sir s;

void df(int x,int nv)
{int i;
 use[x]=1;
 if (nv&1) s[x]=1;
 else s[x]=2;
 for (i=1;i<=n;i++)
   if (G[x][i]&&!use[i])
     df(i,nv+1);
}

```

```

#include <stdio.h>
typedef char mat[26][26];
typedef int sir[26];
mat G; int m,n; sir s;

void solve()
{
  int i,j,k,s;
  for (k=1;k<=n;k++)
    for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
        if (!G[i][j])
          G[i][j]=G[i][k]*G[k][j];
  for (j=1;j<=n;j++) {
    s=0;
    for (i=1;i<=n;i++)
      if (i!=j) s+=G[i][j];
    if (s==n-1) printf("%d ",j);
  }
}

```

```

29.
1 type
2 mat=array[0..25,0..25]of 0..1;
3 sir=array[1..25]of byte;
4 var G:mat; m,n:byte;
5 use:array[1..100]of boolean;
6
7 procedure df(x:byte;
8     var mn,nn:byte);
9 var i:byte;
10 begin
11 use[x]:=true; inc(nn);
12 for i:=1 to n do
13 if (G[x,i]=1) then
14 if not use[i] then begin
15 inc(nn);
16 G[x,i]:=0; G[i,x]:=0;
17 df(i,mn,nn);
18 end else begin
19 G[x,i]:=0; G[i,x]:=0;
20 inc(nn) end
21 end;
22
23 procedure solve();
24 var nr,x,y,i:byte;
25 begin
26 fillchar(use,n,false); nr:=0;
27 for i:=1 to n do
28 if not use[i] then begin
29   x:=0; y:=0; df(i,x,y);
30   nr:=nr + y - x + 1; end;
31 writeln(nr);
32 end;

```

```

37.
1 type
2 mat=array[0..25,0..25]of 0..1;
3 sir=array[1..25]of byte;
4 var G:mat; C,L,T:sir;
5 m,n,x,y:byte;
6 u:array[1..100]of boolean;
7
8 procedure drum(x:byte);
9 begin
10 if x<>0 then begin
11   drum(t[x]); write(x,'') end
12 end;
13
14 procedure BF(nod:integer);
15 var x,i,s,d,mx:integer;
16 begin
17 fillchar(U,sizeof(u),false);
18 s:=1; d:=1; C[s]:=nod; mx:=0;
19 U[nod]:=true; L[nod]:=0;

```

```

#include <stdio.h>
#include <string.h>
typedef char mat[26][26];
typedef int sir[26];
mat G; int m,n,use[101];
void df(int x,int &mn,int &nn){
int i;
use[x]=1; nn++;
for (i=1;i<=n;i++)
if (G[x][i])
if (!use[i]) {
nn++;
G[x][i]=G[i][x]=0;
df(i,mn,nn);
} else {
G[x][i]=G[i][x]=0;
nn++;
}
}

void solve(){
int nr,x,y,i;
memset(use,0,n);
nr=0;
for (i=1;i<=n;i++)
if (!use[i]) {
x=y=0; df(i,x,y);
nr+=y-x+1;
}
printf("%d\n",nr);
}

```

```

#include <stdio.h>
#include <string.h>
typedef char mat[26][26];
typedef int sir[26];
mat G;sir C,L,T;
int m,n,x,y,U[101];

void drum(int x){
if (x!=0) {
drum(T[x]); printf("%d ",x);
}
}

void BF(int nod){
int x,i,s,d,mx;
memset(U,0,sizeof(U));
s=d=1;
C[s]:=nod; U[s]:=1;
mx=0; L[nod]:=0;

```

```

20 while s<=d do begin
21   x:=C[s];
22   for i:=1 to m do begin
23     if not U[i] and (G[x,i]=1)
24     then begin
25       inc(d); C[d]:=i; T[i]:=x;
26       L[i]:=L[x]+1; U[i]:=true;
27       if L[i]>mx then begin
28         mx:=L[i]; y:=i; end;
29     end;
30   end;
31   inc(s);
32 end;
33 writeln(mx);
34 end;

35 begin
36   ... bf(x);
37   ... drum(y);
38 end.

```

```

while (s<=d){
x=C[s];
for (i=1;i<=n;i++)
if ((U[i]&&G[x][i])) {
C[d+1]=i; T[i]=x;
L[i]=L[x]+1; U[i]=1;
if (L[i]>mx){
mx=L[i]; y=i;
}
}
s++;
}
printf("%d\n",mx);
}

int main()
{
BF(x);
drum(y);
return 0;
}

```

### Section 3.1.2

```

16.
1 type e=record
2   h:real; n:string end;
3 var a:array[1..10]of e;
4 sol:array[1..10]of byte;
5 use:array[1..10]of boolean;
6 n,m:integer;
7 .....
8 procedure back(k:integer);
9 var i:integer;
10 begin
11   if k>n then afis
12   else
13     for i:=1 to n do
14       if not use[i] then
15         if (k=1)or(k>1)and
16           (a[i].h>=a[sol[k-1]].h)
17         then begin
18           sol[k]:=i;
19           use[i]:=true;
20           back(k+1);
21           use[i]:=false;
22         end;
23   end;

```

```

18.
1 var i,sum,n,s:longint;
2 sol:array[0..10]of integer;
3 use:array[0..9]of boolean;
4

```

```

#include <stdio.h>
struct e
{
double h;
char n[256];
} a[11];
int sol[11],use[11],n,m;
.....
void back(int k)
{
int i;
if (k>n) afis();
else
for (i=1;i<=n;i++)
if (!use[i])
if (k==1)
(k>1&&a[i].h>=a[sol[k-1]].h))
sol[k]:=i;
use[i]:=1;
back(k+1);
use[i]:=0;
}

```

```

#include <stdio.h>
#include <string.h>
long i,sum,n,s,int sol[11],use[10];

```

```

5 procedure afis(k:byte);
6 begin
7 for i:=1 to k do write(sol[i], '');
8 writeln; end;
9
10 procedure back(k:byte);
11 var i:byte;
12 begin
13 if k<=10 then
14 for i:=sol[k-1]+1 to 9 do
15 if use[i] then
16 if s+i <= sum then begin
17 sol[k]:=i; use[i]:=false;
18 inc(s,i);
19 if s==sum then afis(k);
20 back(k+1);
21 use[i]:=true; dec(s,i);
22 end;
23 end;
24
25 begin
26 readln(n,sum);
27 fillchar(use,10,false);
28 while n>0 do begin
29 use[n mod 10]:=true;
30 n:=n div 10;
31 end;
32 sol[0]:=-1;
33 back(1);
34 end.

```

```

10.
11 var s,i,sum,n:longint;
12 sol,a:array[1..10]of integer;
13
14 procedure afis;
15 begin
16 for i:=1 to n do
17 if sol[i]<>0 then
18 write(sol[i],' ',a[i],'+');
19 writeln;
20 end;
21
22 procedure back(k:byte);
23 var i:byte;
24 begin
25 if k>n then
26 if s==sum then afis
27 else begin
28 for i:=0 to sum div a[k] do
29 if s+a[k]*i<=sum then begin
30 sol[k]:=i; s:=s + i*a[k];
31 back(k+1); s:=s - i*a[k];
32 end;
33 end;

```

```

void afis(int k){
    for (i=1;i<=k;i++)
        printf("%d ",sol[i]);
    printf("\n");
}

void back(int k){
    int i;
    if (k<=10)
        for (i=sol[k-1]+1;i<=9;i++)
            if (use[i])
                if (s+i<=sum){
                    sol[k]:=i;use[i]=0;
                    s+=i;
                    if (s==sum) afis(k);
                    back(k+1);
                    use[i]=1; s-=i;
                }
}

int main(){
    scanf("%ld %ld",&n,&sum);
    memset(use,0,10);
    while (n!=0){
        use[n%10]=1; n/=10;
    }
    sol[0]=-1;
    back(1);
    return 0;
}

```

```

#include <stdio.h>
long s,i,sum,n;
int sol[11],a[11];

procedure afis(){
    int i;
    for (i=1;i<=n;i++)
        if (sol[i]!=0)
            printf("%d%d",sol[i],a[i]);
    printf("\n");
}

void back(int k){
    int i;
    if (k>n)
        if (s==sum) afis();
    else{
        for (i=0;i<=sum/a[k];i++)
            if (s+a[k]*i<=sum){
                sol[k]:=i; s+=i*a[k];
                back(k+1); s-=i*a[k];
            }
    }
}

```

```

19.
1 var i,n:longint;
2 a:array[1..10]of string;
3 sol:array[1..10]of byte;
4 use:array[1..10]of boolean;
5
6 procedure back(k:byte);
7 var i:byte;
8 begin
9 if k>n then afis
10 else
11 for i:=1 to n do
12 if not use[i] then
13 if (k=1)or(k>1)and
14 (abs(i-sol[k-1])>1)then begin
15 sol[k]:=i; use[i]:=true;
16 back(k+1); use[i]:=false;
17 end;
18 end;

```

```

21.
1 const
2 dx:array[1..8]of integer=
3 {-1,-1,-1,0,1,1,1,0};
4 dy:array[1..8]of integer=
5 {-1,0,1,1,1,0,-1,-1};
6 var i,j,m,n,nr:integer;
7 a:array[0..8,0..8]of integer;
8
9 procedure citire;
10 var i,j:integer;
11 begin
12 readln(n,m);
13 for i:=0 to n+1 do
14 for j:=0 to m+1 do a[i,j]:=-1;
15 for i:=1 to n do
16 for j:=1 to m do read(a[i,j]);
17 end;
18
19 procedure fill(x,y,int k);
20 var i:byte;
21 begin
22 if a[x,y]=1 then begin
23 a[x,y]:=k;
24 for i:=1 to 8 do
25 fill(x+dx[i],y+dy[i],k);
26 end; end;
27
28 begin citire; nr:=1;
29 for i:=1 to n do
30 for j:=1 to m do
31 if a[i,j]=1 then begin
32 inc(nr); fill(i,j,nr);
33 end;
34 end.

```

```

#include <stdio.h>
long n,i;
char a[11][256];
int sol[11],use[11];
.
.
.
void back(int k){
    int i;
    if (k>n) afis();
    else
        for (i=1;i<=n;i++)
            if (!use[i])
                if (k==1||(k>1&&
                abs(i-sol[k-1])>1))
                {
                    sol[k]:=i;use[i]=1;
                    back(k+1);use[i]=0;
                }
}

```

```

#include <stdio.h>
const int dx[8]={-1,-1,-1,0,1,1,1,0};
const int dy[8]={-1,0,1,1,1,0,-1,-1};
int i,j,m,n,nr,a[9][9];

```

```

void citire(){
    int i,j;
    scanf("%d %d",&n,&m);
    for (i=0;i<=n+1;i++)
        for (j=0;j<=m+1;j++) a[i][j]=-1;
    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
            scanf("%d",&a[i][j]);
}

```

```

void fill(int x,int y,int k){
    int i;
    if (a[x][y]==1){
        a[x][y]=k;
        for (i=0;i<8;i++)
            fill(x+dx[i],y+dy[i],k);
    }
}

```

```

int main(){
    citire();nr=1;
    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
            if (a[i][j]==1) {
                nr++;
                fill(i,j,nr);
            }
    return 0;
}

```

## Sectiunea 3.2.2

```

1 var n,i,s,p:integer;
2 a:array[0..100] of integer;
3
4 procedure sp(l,r:byte;
5             var s,p:integer);
6 var m,s1,s2,p1,p2:integer;
7 begin
8   if l=r then begin
9     s:=a[l];
10    p:=a[l];
11  end
12 else begin
13   m:=(l+r) div 2;
14   sp(l,m,s1,p1);
15   sp(m+1,r,s2,p2);
16   s:=s1+s2;
17   p:=p1*p2;
18 end;
19 end;
20
21 begin
22   readln(n);
23   for i:=1 to n do read(a[i]);
24   sp(1,n,s,p);
25   writeln(s,' ',p);
26 end.

```

```

1 var n,i,mn,mx:integer;
2 a:array[0..100] of integer;
3 .....
4
5 procedure Mm(l,r:byte;
6             var mn,mx:integer);
7 var m,mn1,mn2,mx1,mx2:integer;
8 begin
9   if l=r then begin
10     mn:=a[l];
11     mx:=a[l];
12   end
13 else begin
14   m:=(l+r) div 2;
15   Mm(l,m,mn1,mx1);
16   Mm(m+1,r,mn2,mx2);
17   if mn1<mn2 then mn:=mn1;
18   else mn:=mn2;
19   if mx1>mx2 then mx:=mx1
20   else mx:=mx2;
21 end;
22 end;

```

```

#include <stdio.h>
int n,i,s,p,a[101];
.....  

void sp(int l,int r,int &s,int &p){
int m,s1,s2,p1,p2;
if (l==r){s=a[l];p=a[l];}
else {
  m=(l+r)/2;
  sp(l,m,s1,p1);
  sp(m+1,r,s2,p2);
  s=s1+s2;
  p=p1*p2;
}
}

int main()
{
  scanf("%d",&n);
  for (i=1;i<=n;i++)
    scanf("%d",&a[i]);
  sp(1,n,s,p);
  printf("%d %d\n",s,p);
  return 0;
}

```

```

#include <stdio.h>
int n,i,mn,mx,a[101];
.....
void Mm(int l,int r,int &mn,
        int &mx)
{
  int m,mn1,mn2,mx1,mx2;
  if (l==r)
    { mn=a[l];
      mx=a[l];
    }
  else
    { m:=(l+r)/2;
      Mm(l,m,mn1,mx1);
      Mm(m+1,r,mn2,mx2);
      if (mn1<mn2) mn=mn1;
      else mn=mn2;
      if (mx1>mx2) mx=mx1;
      else mx=mx2;
    }
}

```

```

3.
4 var n,i,x:integer;
5 a:array[0..100] of integer;
6
7 function Nr(l,r,x:byte):byte;
8 var m:integer;
9 begin
10  if l=r then
11    if a[l]=x then Nr:=1
12    else Nr:=0
13  else begin
14    m:=(l+r) div 2;
15    Nr:=Nr(l,m,x)+Nr(m+1,r,x);
16  end;
17 end;
18

```

```

#include <stdio.h>
int n,i,x,a[101];
.....
int Nr(int l,int r,int x)
{
  int m;
  if (l==r)
    return a[l]==x;
  m:=(l+r)/2;
  Nr(l,m,x)+Nr(m+1,r,x);
}

```

```

19.
20 var n,i,x:integer;
21 a:array[0..100] of integer;
22
23 procedure tai(l,r:byte);
24 var m,i,j:integer;
25 begin
26  if l = r then write(a[l], ' ')
27  else begin
28    m := (l + r) div 2;
29    if odd(r - l + 1) then
30      begin
31        tai(l, m - 1);
32        tai(m + 1, r)
33      end
34    else
35      begin
36        tai(l, m);
37        tai(m + 1, r)
38      end;
39  end;
40 end;
41

```

```

#include <stdio.h>
int n,i,x,a[101];
.....
void tai(int l,int r){
  int m;
  if (l==r)
    printf("%d ",a[l]);
  else{
    m:=(l+r)/2;
    if ((r-l+1)&1) {
      tai(l,m-1);
      tai(m+1,r);
    } else {
      tai(l,m);
      tai(m+1,r);
    }
  }
}

```

```

42.
43 var n,i,x:integer;
44 a:array[0..100] of integer;
45
46 function bin(x:integer):byte;
47 var nr:byte;
48 begin
49  nr:=0;
50  while x<>0 do
51    begin
52      inc(nr,x mod 2);
53      x:=x div 2
54    end;
55  bin:=nr;
56 end;
57

```

```

#include <stdio.h>
int n,i,x,a[101];
int bin(int x)
{
  int nr=0;
  while(x!=0)
  {
    nr+=x&2;
    x/=2;
  }
  return nr;
}

```

```

15 function Nr(l,x,x:byte):byte;
16 var m:integer;
17 begin
18 if l=r then
19 if bin(a[l])=x then Nr:=1
20 else Nr:=0
21 else begin
22 m:=(l+r)div 2;
23 Nr:=Nr(l,m,x)+Nr(m+1,r,x);
24 end;
25 end;

```

```

int Nr(int l,int r,int x)
{
    int m;
    if (l==r)
        return bin(a[l])==x;
    m=(l+r)/2;
    return Nr(l,m,x)+Nr(m+1,r,x);
}

```

### Sectiunea 3.3.2

2. Se va construi vectorul  $S$  în care elementul  $S[k] = \text{True}/1$  dacă se poate obține un premiu de valoare  $k$  și  $\text{False}/0$ , în caz contrar.

```

1 var n:integer;
2 a:array[1..10000] of integer;
3 s:array[0..10000] of boolean;
4 .....
5
6 procedure solve;
7 var i,j,max,nr:integer;
8 begin
9   s[0]:=true;
10  max:=0;
11  nr:=0;
12  for i:=1 to n do begin
13    for j:=max downto 0 do
14      if s[j] and not s[j+a[i]] then begin
15        s[j+a[i]]:=true;
16        inc(nr);
17        if j+a[i]>max then max:=j+a[i];
18      end;
19    end;
20  writeln(nr);
21 end;
22 writeln(nr);
23 end;

```

```

#include <stdio.h>
int n,a[10001];
char s[10001];
.....
void solve()
{
    int i,j,max,nr;
    s[0]=1;
    max=nr=0;
    for (i=1;i<=n;i++)
        for (j=max;j>=0;j--)
            if (s[j]&&!s[j+a[i]])
                {s[j+a[i]]=1;
                 nr++;
                 if (j+a[i]>max)
                     max=j+a[i];
                }
    printf("%d\n",nr);
}

```

3. Se construiesc doi vectori cu următoarele semnificații:

- $A[i]$  = suma maximă a unui subșir cu elemente în primele  $i$  elemente, fără a folosi elementul  $i$ ;
- $B[i]$  = suma maximă a unui subșir cu elemente în primele  $i$  elemente, folosind elementul  $i$ .

Se deduc următoarele relații de recurență:  $A[i] = \max(A[i-1], B[i-1])$ ;  $B[i] = A[i-1] + S[i]$ . Răspunsul va fi  $\max(A[N], B[N])$ .

```

1 intreg n, i, nr, a[0..1], b[0..1];
2 citeste n;
3 pentru i=1,n executa
4   citeste nr;
5   a[1] := max(a[0],b[0]); b[1] := a[0]+nr;
6   a[0] := a[1]; b[0] := b[1];
7
8 scrie max(a[0],b[0]);
9 stop.

```

7. Se construiește un vector  $S$  în care elementul  $S[i] =$  numărul minim de timbre necesare pentru a obține  $i$  centi.

```

1 var n,k:integer;
2 a,s:array[0..10000] of integer;
3
4 procedure solve;
5 var i,j,min,nr:integer;
6 begin
7   s[0]:=0; i:=0;
8   while s[i]<=k do begin
9     inc(i); min:=10000;
10    for j:=1 to n do
11      if (i-a[j])>=0 then
12        if (s[i-a[j]]+1 < min)
13          then min:=s[i-a[j]]+1;
14    s[i]:=min;
15  end;
16  writeln(i-1);
17 end;

```

```

#include <stdio.h>
int n,k,a[10001],s[10001];
void solve()
{
    int i,j,min,nr;
    s[0]=i=0;
    while (s[i]<=k){
        i++;min=10000;
        for (j=1;j<=n;j++)
            if (i-a[j]>=0)
                if (s[i-a[j]]>0)
                    if (s[i-a[j]]+1<min)
                        min=s[i-a[j]]+1;
        s[i]=min;
    }
    printf("%d\n",i-1);
}

```

8. Se determină sirul de numere super prime mai mici decât  $n$ , după care se determină, cu ajutorul programării dinamice, submulțimea de sumă  $n$  cu număr minim de elemente.

Se vor crea doi vectori  $S(0..n)$  și  $T(1..n)$ :

- $S(i)=j$  are semnificația că suma  $i$  se poate obține cu minimum  $j$  termeni;
- $T(i)=k$  înseamnă: suma de valoare  $i$ -a obținut cu ultimul termen  $A(k)$ .

```

1 intreg n,i,j,k,np,p[1000],nr[0..10000],t[0..10000]; logic ok;
2 k ← 0; np ← 0;
3 pentru i←2, n executa
4   ok ← true;
5   pentru j←2,Vi executa
6     daca i mod j=0 atunci ok ← false;
7
8   daca ok=true atunci
9     k ← k+1; ok ← true;
10    pentru j←2,Vk executa
11      daca k mod j=0 atunci ok ← false;
12
13   daca ok=true atunci
14     np ← np + 1;
15     p[np] ← i;
16
17
18

```

## Sectiunea 3.4.2

```

19 nr[0] ← 0;
20 pentru i=1..n executa nr[i] ← ∞;
21
22 pentru i=1..n executa
23   pentru j=0..n-p[i] executa
24     daca nr[j+p[i]] > nr[j]+1 atunci
25       nr[j+p[i]] ← nr[j]+1;
26       t[j+p[i]] ← p[i];
27
28
29 cat timp n>0 executa
30   scrie t[n];
31   n ← n - t[n];
32
33

```

9. Se construiește o matrice  $N \times K$  cu semnificația  $A[i, j] = \text{numărul minim de zile necesare pentru a avea } i \text{ persoane în cameră și } j \text{ perechi de persoane născute în aceeași zi. Există soluție doar dacă } A[N, K] \leq Z. \text{ Relația de recurență va fi:}$

$$A[i, j] = \min(A[i-x, j-x^*(x-1)/2] + 1), x \leq i.$$

Pentru a reconstituia soluția se mai păstrează încă o matrice.

```

1 intreg d, n, z, k, i, j, l, a[50][1225], t[50][1225];
2 d ← 0;
3 pentru i=0..n executa
4   pentru j=0..k executa a[i][j] ← ∞;
5
6
7 a[0][0] ← 0;
8 a[1][0] ← 1;
9 t[1][0] ← 1;
10 pentru i=2..n executa
11   pentru j=1..k executa
12     pentru l=1..i executa
13       daca (j-1*(l-1) div 2 ≥ 0) and (i-l ≥ 0) and
14         (a[i][j] > a[i-1][j-1*(l-1) div 2] + 1) atunci
15           a[i][j] ← a[i-1][j-1*(l-1) div 2] + 1;
16           t[i][j] ← 1;
17
18
19 daca a[n][k] > z atunci scrie 0;
20
21 altfel
22   cat timp n>0 executa
23     i ← t[n][k];
24     pentru j=1..i executa scrie a[n][k];
25     n ← n - i;
26     k ← k - i*(i-1) div 2;
27
28
29
30

```

1. 

```

1 var n:integer;
2 a:array[0..10000] of integer;
3
4 procedure solve;
5 var i,s:integer;
6 begin
7   s:=0;
8   for i:=1 to n do
9     if a[i]>0 then inc(s,a[i]);
10  writeln(s);
11 end;

```

#include <stdio.h>  
int n,a[10001];  
void solve()  
{  
 int i,s=0;  
 for (i=1;i<=n;i++)  
 if (a[i]>0) s+=a[i];  
 printf("%d\n",s);  
}

2. 

```

1 var n:integer;
2 a:array[0..10000] of integer;
3
4 procedure solve;
5 var i,j,x,s:integer;
6 begin
7   for i:=1 to n-1 do
8     for j:=i+1 to n do
9       if a[i]<a[j] then begin
10         x:=a[i];
11         a[i]:=a[j];
12         a[j]:=x;
13       end;
14   s:=0;
15   for i:=1 to n do s:=s+a[i];
16   s:=s div 2;
17   for i:=1 to n do
18     if s-a[i]>0 then begin
19       write(a[i], ' ');
20       s:=s-a[i];
21       a[i]:=0;
22     end;
23   writeln;
24   for i:=1 to n do
25     if a[i]>0 then write(a[i], ' ')
26 end;

```

#include <stdio.h>  
int n,a[10001];  
void solve()  
{  
 int i,j,x,s;  
 for (i=1;i<n;i++)  
 for (j=i+1;j<=n;j++)  
 if (a[i]<a[j]) {  
 x=a[i];  
 a[i]=a[j];  
 a[j]=x;  
 }  
 s=0;  
 for (i=1;i<=n;i++) s+=a[i];
 s/=2;
 for (i=1;i<=n;i++)  
 if (s-a[i]>0){  
 printf("%d ",a[i]);  
 s-=a[i];
 a[i]=0;
 }
 printf("\n");
 for (i=1;i<=n;i++)  
 if (a[i]>0) printf("%d ",a[i]);
}

5. Se parcurg cele două siruri poziție cu poziție, iar unde diferă se face o operație în poziția respectivă. La sfârșit se verifică dacă cele două siruri sunt egale.

```

1 pentru i ← 1..n-k+1 executa
2   daca a[i] ≠ b[i] atunci
3     pentru j ← i..i+k-1 executa
4       a[j] ← -a[j];
5
6     scrie i;
7
8

```

```

7.
1 type v=record s,d:integer end;
2 var a:array[1..100]of v;
3   n:byte;
4
5 procedure solve;
6 var i,j,max,nr,x,y:integer;
7 t:v;
8 begin
9   for i:=1 to n-1 do
10  for j:=i+1 to n do
11    if a[i].s>a[j].s then begin
12      t:=a[i];a[i]:=a[j];a[j]:=t;
13    end;
14  max:=1; nr:=1; x:=a[1].s;
15  y:=a[1].d;
16  for i:=2 to n do
17    if y>=a[i].s then
18      if y>a[i].d then begin
19        y:=a[i].d; inc(nr);
20      end else inc(nr);
21    else begin
22      if nr>max then max:=nr;
23      x:=a[i].s; y:=a[i].d; nr:=1;
24    end;
25  writeln(max);
26 end;

```

```

#include <stdio.h>
struct v {
  int s,d;
} a[101];
int n;

void solve()
{
  int i,j,max,nr,x,y,t;
  for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
      if (a[i].s>a[j].s){
        t=a[i]; a[i]=a[j]; a[j]=t;
      }
  max=nr=1;x=a[1].s;y=a[1].d;
  for (i=2;i<=n;i++)
    if (y>=a[i].s)
      if (y>a[i].d){
        y=a[i].d;nr++;
      } else nr++;
    else {
      if (nr>max) max=nr;
      x=a[i].s; y=a[i].d; nr=1;
    }
  printf("%d\n",max);
}

```

11. Se sortează crescător după limita inferioară a intervalelor. La parcurgerea intervalelor se va actualiza cel mai mare capăt din dreapta (*maxdr*). Orice interval pentru care capătul din dreapta este mai mic decât *maxdr* este redundant, deci inclus în altul.

```

1 type v=record s,d:integer end;
2 var a:array[1..10]of v;
3   n,m:integer;
4
5 procedure solve;
6 var i,j,nr,y:integer;t:v;
7 begin
8   for i:=1 to n-1 do
9     for j:=i+1 to n do
10    if a[i].s>a[j].s then begin
11      t:=a[i];a[i]:=a[j];a[j]:=t;
12    end;
13  nr:=0; y:=a[1].d;
14  for i:=2 to n do
15    if y>a[i].s then
16      if y<a[i].d then
17        y:=a[i].d
18      else inc(nr);
19    else y:=a[i].d;
20  writeln(nr);
21 end;

```

```

#include <stdio.h>
struct v {
  int s,d;
} a[11];
int n,m;

void solve(){
  int i,j,nr,y,t;
  for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
      if (a[i].s>a[j].s){
        t=a[i]; a[i]=a[j]; a[j]=t;
      }
  nr=0;y=a[1].d;
  for (i=2;i<=n;i++)
    if (y>a[i].s)
      if (y<a[i].d) y=a[i].d;
      else nr++;
    else y=a[i].d;
  printf("%d\n",nr);
}

```

