

Ministerul Educației și Cercetării

Vlad Huțanu

Tudor Sorin



Clasa
a XI-a

Manual de
INFORMATICĂ
INTENSIV

Aprobat prin Ordinul MEdC nr. 4446 din 19.06.2006

Editura L&S Soft

Vlad Huțanu

Tudor Sorin

Huțanu

INFORMATICĂ INTENSIV

(filiera *teoretică*, profilul *real*, specializarea
matematică-informatică, intensiv informatică)

Ciclul superior al liceului,
clasa a XI-a

Editura L&S Soft
București

Toate drepturile asupra acestei lucrări aparțin editurii L&S SOFT.
Reproducerea integrală sau parțială a textului din această carte este posibilă
doar cu acordul în scris al editurii L&S SOFT.

Manualul a fost aprobat prin Ordinul ministrului Educației și Cercetării nr. 4446 din 19.06.2006 în urma evaluării calitative organizate de către Consiliul Național pentru Evaluarea și Difuzarea Manualelor și este realizat în conformitate cu programa analitică aprobată prin Ordin al ministrului Educației și Cercetării nr. 3252 din 13.02.2006.

Referenți științifici:

Prof. Dr. Victor Mitrana, Facultatea de Matematică, Universitatea București
Prof. grad I Valiana Petrișor, Colegiul Național Bilingv George Coșbuc

Tiparul executat la S.C. LUMINATIPO s.r.l.
Str. Luigi Galvani nr. 20 bis, sector 2, București

Anul tipăririi: 2006

Descrierea CIP a Bibliotecii Naționale a României

HUȚANU, VLAD

Informatică intensiv : manual pentru ciclul superior al liceului : clasa a XI-a - (filiera teoretică, profilul real, specializarea matematică-informatică, intensiv informatică) / Vlad Huțanu, Sorin Tudor. - București : Editura L & S Soft, 2006

ISBN (10) 973-88037-0-5; ISBN (13) 978-973-88037-0-1

I. Tudor, Sorin

004(075.35)

Editura L&S SOFT:



Adresa: Str. Stânjeneilor 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.

Cuprins

Capitolul 1. Alocarea dinamică a memoriei.....	7
1.1. Generalități	7
1.2. Variabile de tip pointer.....	8
1.2.1. Variabile de tip pointer în Pascal.....	8
1.2.2. Variabile de tip pointer în C++.....	11
1.3. Alocarea dinamică a memoriei.....	14
1.3.1. Alocarea dinamică în Pascal.....	14
1.3.2. Alocarea dinamică în C++.....	17
Probleme propuse.....	21
Răspunsuri	22
 Capitolul 2. Liste liniare	 23
2.1. Definiția listelor	23
2.2. Liste liniare alocate simplu înlántuit.....	24
2.2.1. Prezentare generală.....	24
2.2.2. Crearea și afișarea listelor.....	24
2.2.3. Operații asupra unei liste liniare.....	28
2.2.4. Aplicații ale listelor liniare.....	34
2.2.4.1. Sortarea prin inserție.....	34
2.2.4.2. Sortarea topologică.....	36
2.2.4.3. Operații cu polinoame.....	41
2.2.5. Liste liniare alocate dublu înlántuit.....	50
2.2.5.1. Crearea unei liste liniare alocate dublu înlántuit.....	50
2.2.5.2. Adăugarea unei înregistrări la dreapta.....	51
2.2.5.3. Adăugarea unei înregistrări la stânga.....	51
2.2.5.4. Adăugarea unei înregistrări în interiorul listei.....	51
2.2.5.5. Ștergerea unei înregistrări din interiorul listei.....	52
2.2.5.6. Ștergerea unei înregistrări la stânga/dreapta listei.....	53
2.2.5.7. Listarea de la stânga la dreapta listei.....	53
2.2.5.8. Listarea de la dreapta la stânga listei	53
2.2.6. Stiva implementată ca listă liniară simplu înlántuită.....	55
2.2.7. Coada implementată ca listă liniară simplu înlántuită.....	56
Probleme propuse.....	58
Răspunsuri la testele grilă	63
 Capitolul 3. Metoda Divide et Impera.....	 64
3.1. Prezentare generală.....	64
3.2. Aplicații.....	64
3.2.1. Valoarea maximă dintr-un vector.....	64
3.2.2. Sortarea prin interclasare.....	66
3.2.3. Sortarea rapidă.....	68
3.2.4. Turnurile din Hanoi.....	71
3.2.5. Problema tăieturilor.....	72

3.3. Fractali.....	75
3.3.1. Elemente de grafică.....	75
3.3.1.1. Generalități (varianta Pascal).....	75
3.3.1.2. Generalități (varianta C++).....	77
3.3.1.3. Setarea culorilor și procesul de desenare (Pascal și C++).....	78
3.3.2. Curba lui Koch pentru un triunghi echilateral.....	80
3.3.3. Curba lui Koch pentru un pătrat.....	83
3.3.4. Arboarele.....	85
Probleme propuse.....	87
Răspunsuri.....	88
 Capitolul 4. Metoda Backtracking.....	90
4.1. Prezentarea metodei.....	90
4.1.1. Când se utilizează metoda backtracking ?.....	90
4.1.2. Prințipiu care stă la baza metodei backtracking.....	90
4.1.3. O modalitate de implementare a metodei backtracking.....	92
4.1.4. Problema celor n dame.....	95
4.2. Mai puține linii în programul sursă.....	98
4.3. Cazul în care se cere o singură soluție. Exemplificare: problema colorării hărților.....	101
4.4. Aplicații ale metodei backtracking în combinatorică.....	103
4.4.1. O generalizare utilă.....	103
4.4.2. Produs cartezian.....	104
4.4.3. Generarea tuturor submulțimilor unei mulțimi.....	106
4.4.4. Generarea combinărilor.....	108
4.4.5. Generarea aranjamentelor.....	110
4.4.6. Generarea tuturor partițiilor mulțimii {1, 2, ..., n}.....	112
4.5. Alte tipuri de probleme care se rezolvă prin utilizarea metodei backtracking.....	114
4.5.1. Generalități.....	114
4.5.2. Generarea partițiilor unui număr natural.....	115
4.5.3. Plata unei sume cu bancnote de valori date.....	117
4.5.4. Problema labirintului.....	119
4.5.5. Problema bilei.....	122
4.5.6. Săritura calului.....	124
Probleme propuse.....	125
Indicații.....	128
 Capitolul 5. Metoda Greedy	129
5.1. Generalități	129
5.2. Probleme pentru care metoda Greedy conduce la soluția optimă.....	130
5.2.1. Suma maximă.....	130
5.2.2. Problema planificării spectacolelor.....	131
5.2.3. Problema rucsacului (cazul continuu).....	133
5.2.4. O problemă de maxim.....	135
5.3. Greedy euristic.....	137
5.3.1. Plata unei sume într-un număr minim de bancnote.....	137
5.3.2. Săritura calului.....	139
5.3.3. Problema comis-voiajorului.....	141
Probleme propuse.....	142
Răspunsuri / Indicații.....	144

 Capitolul 6. Programare dinamică	145
6.1. Generalități	145
6.2. Problema triunghiului.....	147
6.3. Subșir crescător de lungime maximă.....	151
6.4. O problemă cu sume.....	154
6.5. Problema rucsacului (cazul discret).....	156
6.6. Distanța Levenshtein.....	161
6.7. Înmulțirea optimă a unui sir de matrice.....	167
6.8. Probleme cu ordinea lexicografică a permutărilor.....	174
6.9. Numărul partițiilor unei mulțimi cu n elemente.....	177
Probleme propuse.....	180
Indicații.....	183
 Capitolul 7. Grafuri neorientate	186
7.1. Introducere	186
7.2. Definiția grafului neorientat.....	187
7.3. Memorarea grafurilor.....	188
7.4. Graf complet.....	196
7.5. Graf parțial, subgraf.....	197
7.6. Parcurgerea grafurilor neorientate.....	198
7.6.1. Parcurgerea în lățime (BF – Breadth First).....	199
7.6.2. Parcurgerea în adâncime (DF – Depth First).....	201
7.6.3. Estimarea timpului necesar parcurgerii grafurilor.....	203
7.7. Lanțuri.....	203
7.8. Graf conex.....	207
7.9. Componente conexe.....	208
7.10. Cicluri.....	210
7.11. Ciclu eulerian, graf eulerian.....	212
7.12. Grafuri bipartite.....	215
7.13. Grafuri hemiltoniene.....	217
Probleme propuse.....	221
Răspunsuri.....	227
 Capitolul 8. Grafuri orientate	230
8.1. Noțiunea de graf orientat.....	230
8.2. Memorarea grafurilor orientate.....	233
8.3. Graf parțial, subgraf.....	238
8.4. Parcurgerea grafurilor. Drumuri. Circuite.....	239
8.5. Graf complet și graf turneu.....	241
8.6. Graf tare conex. Componente tare conexe.....	243
8.7. Drumuri de cost minim.....	246
8.7.1. Introducere.....	246
8.7.2. Algoritmul Roy-Floyd.....	247
8.7.3. Utilizarea algoritmul Roy-Floyd pentru determinarea drumurilor de cost maxim.....	251
8.7.4. Algoritmul lui Dijkstra.....	252
Probleme propuse.....	256
Răspunsuri.....	260

Capitolul 9. Arbori	261
9.1. Noțiunea de arbore.....	261
9.2. Noțiunea de arbore parțial.....	263
9.3. Mai mult despre cicluri.....	264
9.4. Arbori cu rădăcină.....	267
9.4.1. Noțiunea de arbore cu rădăcină.....	267
9.4.2. Memorarea arborilor cu rădăcină prin utilizarea referințelor descendente.....	268
9.4.3. Memorarea arborilor cu rădăcină prin utilizarea referințelor ascendentе.....	268
9.4.4. Înălțimea unui arbore cu rădăcină.....	270
9.5. Noțiunea de pădure.....	272
9.6. Arbori parțiali de cost minim.....	275
9.6.1. Algoritmul lui Kruskal.....	276
9.6.2. Algoritmul lui Prim.....	280
9.7. Arbori binari.....	285
9.7.1. Noțiunea de arbore binar. Proprietăți.....	285
9.7.2. Modalități de reprezentare a arborilor binari.....	287
9.7.3. Modalități de parcursere a arborilor binari.....	287
9.7.4. O aplicație a arborilor binari: forma poloneză a expresiilor.....	291
9.7.5. Arbore binar complet.....	297
9.7.6. MinHeap-uri și MaxHeap-uri. Aplicații.....	299
9.7.7. Arbori de căutare.....	304
Probleme propuse.....	310
Răspunsuri.....	315
Anexa 1. Aplicații practice ale grafurilor.....	316

Alocarea dinamică a memoriei

1.1. Generalități

După cum știți, fiecărui program î se alocă trei zone distincte în memoria internă, zone în care se găsesc memorate variabilele programului. În acest capitol, vom învăța să alocăm variabile în **Heap**. De asemenea, vom învăța să accesăm conținuturile variabilelor, atât cele din segmentul de date, cât și cele din **Heap**, pornind de la adresa lor din memorie.

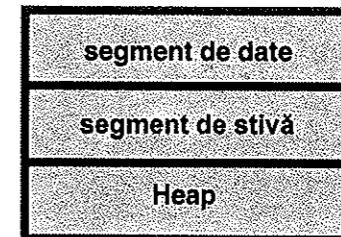


Figura 1.1. Cele trei zone din memoria internă



Definiția 1.1. Prin **pointer** înțelegem adresa unei variabile, iar printr-o variabilă de tip pointer vom înțelege o variabilă care poate reține adresele altor variabile.



Definiția 1.2. Prin **alocarea dinamică** a memoriei vom înțelege alocarea unor variabile în **Heap**, alocare care **se face în timpul executării programului** și nu de la început, aşa cum am fost obișnuiți până acum. Mecanismul alocării dinamice a memoriei necesită utilizarea pointerilor și evident, a variabilelor de tip pointer.

Avantajele alocării dinamice sunt:

- ⦿ Programul va utiliza atâta memorie cât are nevoie. Nu întotdeauna se poate cunoaște, de la început, de câtă memorie are nevoie, aceasta se decide în timpul executării, în funcție de datele de intrare. În plus, dacă o variabilă nu mai este necesară, există posibilitatea să eliberăm memoria ocupată de aceasta. O astfel de abordare conduce la micșorarea substanțială a necesarului de memorie a unui program.

- ⇒ În varianta Borland a limbajelor Pascal și C++, memoria din segmentul de date nu este întotdeauna suficientă, ea este limitată la **64K**. Apelând la **Heap**, se mărește memoria disponibilă.¹
- ⇒ Anumite structuri de date, pe care le vom studia în amănunt, se implementează cu ușurință în **Heap**. Un exemplu în acest sens sunt listele liniare, dar acestea sunt prezentate în capitolul următor.

1.2. Variabile de tip pointer

1.2.1. Variabile de tip pointer în Pascal

Am învățat faptul că memoria internă poate fi privită ca o succesiune de octeți. Pentru a-i distinge, aceștia sunt numerotati.



Definiția 1.3. Numărul de ordine al unui octet se numește **adresa** lui.

Orice variabilă ocupă un număr de octeți succesivi. De exemplu, o variabilă de tip **integer** ocupă doi octeți.



Definiția 1.4. Adresa primului octet al variabilei se numește **adresa variabilei**.



Observație. Nu trebuie confundată adresa unei variabile cu valoarea pe care aceasta o memorează!

Memorarea adreselor variabilelor se face cu ajutorul variabilelor de tip **pointer**.

- ⇒ Variabilele de tip pointer se caracterizează prin faptul că valorile pe care le pot memora sunt adrese ale altor variabile.
- ⇒ Ele **nu** pot fi citite, **nu** pot fi tipărite în mod direct și, cu o excepție, conținutul lor **nu** poate fi modificat în urma unor operații aritmétice (de exemplu, nu putem incrementa o astfel de variabilă).

Limbajul Pascal face **discriminare** între natura adreselor care pot fi memorate. Astfel, există adrese ale variabilelor de tip **integer** (formând un tip identificat prin sintagma uzuală **"pointer către variabile de tip integer"**), adrese ale variabilelor de tip **real** ("pointer către variabile de tip **real**"), adrese ale variabilelor de tip **string** ("pointer către variabile de tip **string**"). Din acest motiv, tipul **pointer** este variat.

¹ În ultimele versiuni ale celor două limbaje, memoria din segmentul de date nu mai este limitată, ci se poate folosi întreaga memorie disponibilă. Din acest punct de vedere, dacă se folosesc aceste versiuni, avantajul "memoriei în plus" nu mai poate fi luat în considerare.

- ⇒ Tipul unei astfel de variabile **pointer** se declară ca mai jos:

```
type nume=^tip;
```

- Ex:** 1. Variabile de tip pointer către variabile de tip **integer**. Variabilele **adr1**, **adr2**, pot reține adrese ale variabilelor de tip **intreg**.

```
type adr_int=^integer;
var adr1,adr2:adr_int;
    numar:integer;
```

- Ex:** 2. Variabile de tip pointer către variabile de tip **real**. Variabila **adresa**, poate reține adrese ale variabilelor de tip **real**.

```
type adr_real=^real;
var adresa: adr_real;
```

- Ex:** 3. Variabile de tip pointer către variabile de tip **inreg**. Tipul **inreg** este tip **record**. Variabila **adr_inr**, poate reține adrese ale variabilelor de tipul **inreg**.

```
type inreg=record
    nume:string[10];
    prenume:string[10];
    varsta:byte;
end;

type adr_inreg=^inreg;
var adr_inr:adr_inreg;
```

Observații

- ✓ Între variabilele de tip pointer sunt permise atribuiri doar în cazul în care au același tip pointer (rețin adrese către același tip de variabile).

Exemplu:

```
adr1, adr2:^integer;
adr3: ^real;
```

Atribuirea **adr1:=adr2** este corectă, iar atribuirea **adr3:=adr2** nu este corectă.

- ✓ Chiar și în cazul în care avem tipurile identice, dar descrise diferit, atribuirea nu este posibilă. Atribuirea din exemplul de mai jos este eronată.

Exemplu:

```
type adrese=^integer;
var adr1:adrese;
    adr2:^integer;
...
adr2:=adr1;
```

- ⇒ Pentru a obține adresa unei variabile oarecare se folosește operatorul “@” prefixat. Adresa va fi memorată de o variabilă pointer către tipul variabilei a cărei adresă a fost returnată de operatorul “@”.
- ⇒ Pornind de la o variabilă de tip pointer care memorează adresa unei variabile, cu ajutorul operatorului “^”, postfixat, se poate adresa conținutul variabilei a cărei adresă este memorată.

Ex: 1. În programul următor, variabilei **x**, care este de tip **integer**, î se atribuie valoarea **3**. Variabilei **a**, de tip pointer către **integer**, î se atribuie adresa lui **x**.

Pornind de la conținutul variabilei **a**, se afișează conținutul variabilei **x**:

```
type adrint:^integer;
var a:adrint;
    x:integer;
begin
    x:=3;
    a:=@x;
    writeln(a^);
end.
```

Ex: 2. În programul următor, variabilei **v**, de tip **Inreg**, î se atribuie o valoare. Variabilei **adresa**, de tip pointer către **Inreg**, î se atribuie adresa lui **v**.

Pornind de la conținutul variabilei **adresa**, se afișează conținutul variabilei **v**.

```
type adrInreg.^inreg;
    inreg=record
        nume:string;
        varsta:integer;
    end;

var v:inreg;
    adresa:adrInreg;

begin
    v.nume:='Popescu';
    v.varsta:=17;
    adresa:=@v;
    writeln(adresa^.nume, ' ', adresa^.varsta);
end.
```

1.2.2. Variabile de tip pointer în C++

Am învățat faptul că memoria internă poate fi privită ca o succesiune de octeți. Pentru a-i distinge, aceștia sunt numerotați.



Definiția 1.3. Numărul de ordine al unui octet se numește **adresa** lui.

Orice variabilă ocupă un număr de octeți succesivi. De exemplu, o variabilă de tip **int** ocupă doi octeți (în varianta **Borland C++ 3.0**).



Definiția 1.4. Adresa primului octet al variabilei se numește **adresa variabilei**.



Observații

- ✓ Nu trebuie confundată adresa unei variabile cu valoarea pe care aceasta o memorează!
- ✓ Uneori, în loc de adresă a unei variabile vom folosi termenul **pointer**!

Memorarea adreselor variabilelor se face cu ajutorul variabilelor de tip **pointer**.

- ⇒ Variabilele de tip pointer se caracterizează prin faptul că valorile pe care le pot memora sunt adrese ale altor variabile.

Limbajul **C++** face **distincție** între natura adreselor care pot fi memorate. Astfel, există adrese ale variabilelor de tip **int**, adrese ale variabilelor de tip **float**, adrese ale variabilelor de tip **char**, etc. Din acest motiv și tipul variabilelor de tip pointer este diferit.

- ⇒ Tipul unei variabile de tip pointer se declară ca mai jos:

tip *nume.

Ex: 1. Variabile de tip pointer către variabile de tip **int**. Variabilele **adri1** și **adri2** pot reține adrese ale variabilelor de tip **int**. Priviți declarația de mai jos:

```
int *adri1, *adri2;
```

Ex: 2. Variabile de tip pointer către variabile de tip **float**. Variabila **adresa**, poate reține adrese ale variabilelor de tip **float**:

```
float* adresa;
```

Ex: 3. Variabile de tip pointer către variabile de tip **elev**, care la rândul lor sunt de tip **struct**. Variabilele **a** și **b**, pot reține adrese ale variabilelor de tipul **elev**.

```
struct elev
{ char nume[20], prenume[20];
  float nota_mate, nota_info;
  int varsta;
};

elev *a,*b;
```

! Observații

- ✓ Caracterul "*" poate fi așezat în mai multe feluri, după cum se observă:

```
int* adr1; int * adr1; int *adr1;
```

- ✓ Pentru a declara mai multe variabile de acest tip, caracterul "*" se trece de fiecare dată:

```
int *adr1, *adr2, *adr3;
```

- ✓ O declarație de genul "int* adr1, adr2;" are semnificația că **adr1** este de tip pointer către **int**, în vreme ce **adr2** este de tip **int**.

Atenție! Aici se greșește deseori...

- ⇒ Adresa unei variabile se obține cu ajutorul **operatorului de referențiere "&"**, care trebuie să preceadă numele variabilei:

```
&Nume_variabila;
```

Ex: **adr1=&numar;** - variabilei **adr1** i se atribuie adresa variabilei **numar**.

- ⇒ Fiind dată o variabilă de tip pointer către variabile de un anume tip, care memorează o adresă a unei variabile de acel tip, pentru a obține conținutul variabilei a cărei adresă este memorată, se utilizează operatorul unar "*", numit și **operator de derefențiere**.

Ex: 1. Variabila **a** este inițializată cu 7, iar variabila **adr** este inițializată cu adresa lui **a**. Secvența afișează conținutul variabilei **a** (7), pornind de la adresa ei, reținută de **adr**:

```
int a=7, *adr=&a;
cout<<*adr;
```

Ex: 2. Variabila **a** de tip **elev** este inițializată, iar variabila **adra**, de tip pointer către variabile de tip **elev** este inițializată cu adresa variabilei **a**. Secvența următoare tipărește conținutul variabilei **a**:

```
...
struct elev
{
  char nume[20], prenume[20];
};

...
elev a, *adra=&a;
strcpy(a.nume, "Bojian");
strcpy(a.prenume, "Andronache");
cout<<(*adra).nume<< " <<(*adra).prenume<<endl;
```

! Observați modul în care am obținut conținutul unui câmp al variabilei **a**, pornind de la pointerul către **a**:

```
(*adra).nume.
```

De ce este nevoie de paranteze?

Operatorul **"."** - numit **operator de selecție**, are prioritatea 1, deci maximă.

Operatorul **"*"** - unar, numit și **operator de derefențiere**, are prioritatea 2, mai mică. Prin urmare, în absența parantezelor rotunde, se încearcă mai întâi evaluarea expresiei "**adra.nume**", expresie care n-are sens! Parantezele schimbă ordinea de evaluare, se evaluatează mai întâi "***adra**", expresie care are sens.

⇒ Pentru o astfel de selecție, în loc să folosim trei operatori, se poate utiliza unul singur, **operatorul de selecție indirectă**: **"->"**. Acesta accesază un câmp al unei structuri pornind de la un pointer (adresă) către acea structură. El are prioritatea maximă - vezi tabelul operatorilor.

Tipărirea se poate face și așa:

```
cout<<adra->nume<< " <<adra->prenume;
```

Între variabile de tip pointer sunt permise atribuiri doar în cazul în care au același tip pointer (rețin adrese către același tip de variabile).

Exemplu:

```
int *adr1, *adr2;
float *adr3;
... // initializari
```

Atribuirea "**adr1=adr2**" este corectă, iar atribuirea "**adr3=adr2**" nu este corectă.

? În aceste condiții, vă puteți întreba: cum putem atribui conținutul unei variabile de tip pointer către tipul **x**, altei variabile de tip pointer către tipul **y**? În definitiv, amândouă rețin o adresă... În acest caz, se utilizează **operatorul de conversie explicită**. De această dată, pentru exemplul anterior, atribuirea: "**adr3=(float*)adr2**" este corectă.

1.3. Alocarea dinamică a memoriei

1.3.1. Alocarea dinamică în Pascal

Anumite variabile pot fi **alocate dinamic**. Asta înseamnă că:

- Spațiul necesar memorării este rezervat într-un segment special destinat acestui scop, numit **HEAP**.
 - Rezervarea spațiului se face în timpul executării programului, atunci când se execută o anumită procedură, scrisă special în acest scop.
 - Atunci când variabila respectivă nu mai este utilă, spațiul din memorie este eliberat, pentru a fi rezervat, dacă este cazul, pentru alte variabile.
- În **Borland Pascal**, pentru alocarea dinamică se utilizează următoarele două proceduri:

- Procedura **New** alocă spațiu în **HEAP** pentru o variabilă dinamică. După alocare, adresa variabilei se găsește în **P**¹.

```
procedure New(var P: Pointer)
```

- Procedura **Dispose** eliberează spațul rezervat pentru variabila a cărei adresă este reținută în **P**. După eliberare, conținutul variabilei **P** este nedefinit.

```
procedure Dispose(var P: Pointer)
```

Mecanismul alocării dinamice este următorul:

- Se declară o variabilă pointer, s-o numim **P**, care permite memorarea unei adrese.
- Se alocă variabila dinamică prin procedura **New**, de parametru **P**. În urma alocării, variabila **P** reține adresa variabilei alocate dinamic.
- Orice acces la variabila alocate dinamic se face prin intermediul variabilei **P**.

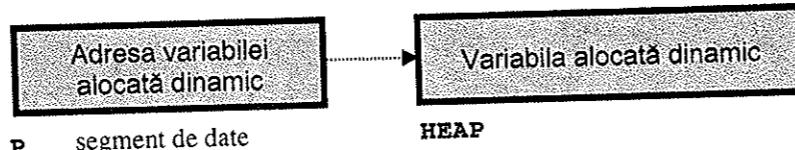


Figura 1.2. Accesul la variabila alocate dinamic

¹ Dacă nu există spațiu în **HEAP**, **P** reține **nil** (nici o valoare). În practică, întotdeauna se face testul existenței spațiului. Din motive didactice, pentru a nu complica programele, în exemplele pe care le vom da nu vom testa existența spațiului în **HEAP**.

⇒ Pentru a elibera spațiul ocupat de o variabilă dinamică, a cărei adresă se găsește în **P**, se utilizează **Dispose**, de parametru **P**. După eliberarea spațiului rezervat pentru variabila dată, conținutul variabilei **P** este nedefinit.

⇒ Fiind dată o variabilă de tip pointer către variabile de un anumit tip, pentru a accesa conținutul variabilei a cărei adresă este memorată, se utilizează numele variabilei de tip pointer urmat de operatorul "**^**".

Ex: 1. Variabile de tip pointer către variabile de tip **integer**. Variabilele **adr1** și **adr2** pot reține adrese ale variabilelor de tip întreg.

```
type adr_int=^integer;
var adr1:adr_int;
{sau adr1:^integer prin renuntarea la prima linie}
...
new(adr1) {alloc spatiu in HEAP pentru o variabila de tip
          intreg}
adr1^:=7; {variabila retine 7}
writeln(adr^) {tiparesc continutul acestei variabile (7)}
dispose(adr) {eliberez spatiul}
```

Ex: 2. Variabile de tip pointer către variabile de tip **real**. Variabila **adresa**, poate reține adrese ale variabilelor de tip **real**.

```
type adr_real=^real;
var adresa: adr_real;
...
new(adresa);
adresa^:=7.65;
writeln(adresa^:4:2);
```

Ex: 3. Variabile de tip pointer către variabile de tip **inreg**, care la rândul lor, sunt de tip **record**. Variabila **adr_inr**, poate reține adrese ale variabilelor de tipul **inreg**.

```
type inreg=record
  nume:string[10];
  prenume:string[10];
  varsta:byte;
end;
adr_inreg=^inreg;

var adr:adr_inreg;

begin
  readln(adr^.nume);
  readln(adr^.prenume);
  readln(adr^.varsta);
  writeln(adr^.nume);
  writeln(adr^.prenume);
  writeln(adr^.varsta);
  ...
```

Ex: 4. Programul următor citește și afișează o matrice. Nou este faptul că matricea este alocată în **HEAP**.

```
type matrice=array[1..10,1..10]of integer;
  adr_mat^matrice;
var adr:adr_mat;
  m,n,i,j:integer;

begin
  write('m=');readln(m);
  write('n=');readln(n);
  new(adr);
  for i:=1 to m do
    for j:=1 to n do
      readln(adr^[i,j]);
  for i:=1 to m do
  begin
    for j:=1 to n do write(adr^[i,j]:4);
    writeln;
  end;
end.
```

Ex: 5. Este cunoscut faptul că funcțiile nu pot întoarce decât tipuri simple. Prin urmare, o funcție nu poate întoarce o matrice care este descrisă de un tip structurat. Dar tipul pointer este simplu. Aceasta înseamnă că o funcție poate întoarce un pointer. În cazul în care avem un pointer către o matrice (reținută în **HEAP**) se poate spune, prin abuz de limbaj, că o funcție întoarce o matrice. Programul următor citește două matrice și afișează suma lor. Matricele sunt rezervate în **HEAP**.

```
type matrice=array[1..10,1..10]of integer;
  adr_mat^matrice;
var adr1,adr2,adr3:adr_mat;
  m,n:integer;

function Cit_Mat(m,n:integer):adr_mat;
var i,j:integer;
  adr:adr_mat;
begin
  new(adr);
  for i:=1 to m do
    for j:=1 to n do  readln(adr^[i,j]);
  Cit_Mat:=adr;
end;

function Suma_Mat(adr1,adr2:adr_mat):adr_mat;
var i,j:integer;
  adr:adr_mat;

begin
new(adr);
for i:=1 to m do
  for j:=1 to n do adr^[i,j]:=adr1^[i,j]+adr2^[i,j];
  Suma_Mat:=adr;
end;
```

```
procedure Tip_Mat(adr:adr_mat);
var i,j:integer;
begin
  for i:=1 to m do
  begin
    for j:=1 to n do write(adr^[i,j]:4);
    writeln;
  end;
end;

begin
  write('m='); readln(m);
  write('n='); readln(n);
  adr1:=Cit_Mat(m,n); adr2:=Cit_Mat(m,n);
  adr3:=Suma_Mat(adr1,adr2);
  Tip_Mat(adr3);
end.
```

1.3.2. Alocarea dinamică în C++

În C++, pentru alocarea dinamică se utilizează următorii operatori:

⇒ Operatorul **new** alocă spațiu în **HEAP** pentru o variabilă dinamică. După alocare, adresa variabilei se atribuie lui **P**, unde **P** este o variabilă de tip pointer către tip¹:

P=new tip.

Observații

- ✓ Numărul de octeți alocati în **HEAP** este, evident, egal cu numărul de octeți ocupat de o variabilă de tipul respectiv.
- ✓ Durata de viață a unei variabile alocate în **HEAP** este până la eliberarea spațiului ocupat (cu **delete**) sau până la sfârșitul executării programului.
- ⇒ Operatorul **delete** eliberează spațiul rezervat pentru variabila a cărei adresă este reținută în **P**. După eliberare, conținutul variabilei **P** este nedefinit.

delete P.

Ex: 1. Variabile de tip pointer către variabile de tip **int**. Variabila **adr1** poate reține adrese ale variabilelor de tip **int**.

```
...
int* adr1;
adr1=new int; // aloc spatiu in HEAP pentru o var. de tip int
*adr1=7; //variabila alocata retine 7
cout<<*adr1; // tiparesc continutul variabilei
delete adr1; // eliberez spatiul
```

¹ Dacă nu există spațiu în **HEAP**, **P** reține 0 (nici o valoare). În practică, întotdeauna se face testul existenței spațiului. Din motive didactice, pentru a nu complica programele, în exemplele pe care le vom da nu vom testa existența spațiului în **HEAP**.

Ex: 2. Variabile de tip pointer către variabile de tip **float**. Variabila **adresa**, poate reține adrese ale variabilelor de tip **float**.

```
float* adresa; // sau float* adresa=new float;
adresa=new float;
*adresa=7.65;
cout<<*adresa;
```

Ex: 3. Variabile de tip pointer către variabile de tip **inreg**, care la rândul lor, sunt de tip **struct**. Variabila **adr_inr**, poate reține adrese ale variabilelor de tipul **inreg**.

```
#include <iostream.h>
struct inreg
{
    char nume[20], prenume[20];
    int varsta;
};
main()
{
    inreg* adr;
    adr=new inreg;
    cin>>adr->nume>>adr->prenume>>adr->varsta;
    cout<<adr->nume<<endl<<adr->prenume<<endl<<adr->varsta;
}
```

Mecanismul alocării dinamice

În continuare prezentăm mecanismul alocării dinamice a masivelor. Pentru început, vom prezenta pe scurt legătura între pointeri și masive.

⇒ Un tablou **p**-dimensional se declară astfel:

tip nume[n₁] [n₂]...[n_p]

Exemple

- **float A[7][4][2];** - am declarat un tablou cu 3 dimensiuni, unde tipul de bază este **float**.
- **long b[9][7][8][5];** - am declarat un tablou cu 4 dimensiuni, unde tipul de bază este **long**.

⇒ *Numele tabloului **p** dimensional de mai sus este pointer constant către un tablou **p-1** dimensional de forma [n₂]...[n_p], care are componente de bază de același tip cu cele ale tabloului. Exemplile de mai jos se referă la tablourile anterior prezентate:*

- **A** este pointer constant către tablouri cu [4][2] componente de tip **float**;
- **b** este pointer constant către tablouri cu [7][8][5] componente de tip **long**.

⇒ Un pointer către un tablou **k** dimensional cu [l₁][l₂]...[l_k] componente de un anumit tip se declară astfel:

tip (*nume)[l₁] [l₂]... [l_k].

Exemple:

- variabila de tip pointer, numită **p**, care poate reține pe **A**, se declară:

float (*p)[4][2];

- variabila de tip pointer, numită **q**, care poate reține pe **b**, se declară:

long (*q)[7][8][5];

? De ce se folosesc parantezele rotunde? Operatorul de tip **array ([])** are cea mai mare prioritate (mai mare decât operatorul **"*"**). Declarația:

float *p[7][8][5];

este interpretată ca masiv cu [7][8][5] componente de tip **float***. În concluzie, este masiv cu componente de tip pointer și nu pointer către masive. **Atenție!** Aici se fac multe confuzii...

⇒ Întrucât numele unui masiv **p** dimensional este pointer către un masiv **p-1** dimensional, pentru a aloca dinamic un masiv se va utiliza un pointer către masive **p-1** dimensionale (ultimele **p-1** dimensiuni).

Ex: 1. Alocăm în **HEAP** un vector cu 4 componente de tip **int**. Numele unui astfel de vector are tipul **int*** (pointer către **int**). Prin urmare, variabila **a** are tipul **int***. După alocare, ea va conține adresa primului element al vectorului din **HEAP**. Din acest moment, pornind de la pointer (reținut în **a**) vectorul se adresează exact cum suntem obișnuiți.

```
int *a =new int[4];
a[3]=2;
```

! Observați modul în care a fost trecut tipul în dreapta operatorului **new**. Practic, declararea tipului se face întocmai ca declarația masivului, însă numele a fost eliminat.

Ex: 2. Declarăm în **HEAP** o matrice (masiv bidimensional) cu 3 linii și 5 coloane, cu elemente de tip **double**:

```
double (*a)[5]=new double [3][5];
a[1][2]=7.8;
```

⇒ În cazul masivelor, trebuie atenție la eliberarea memoriei (cu **delete**). Trebuie ținut cont de tipul pointerului, aşa cum rezultă din exemplele următoare.

Ex: 1. Fie vectorul alocat în **HEAP**, **int *a =new int[4];**. Dacă încercăm dezalocarea sa prin: **delete a;**, îi dezalocăm prima componentă, pentru că pointerul este de tip **int***.

Corect, dezalocarea se poate face prin: "**delete [4] a;**" - eliberăm spațiul pentru toate componente (în acest caz avem 4). Observați că operatorul **delete** se poate utiliza și ca mai sus.

Ex: 2. Fie matricea alocată în **HEAP**:

```
double (*a)[5]=new double [3][5];
```

Eliberarea spațiului ocupat de ea se face prin "delete [3] a;".

□ **Aplicația 1.1.** Programul următor citește și afișează o matrice. Nou este faptul că matricea este alocată în **HEAP**.

```
#include <iostream.h>
main()
{ int m,n,i,j,(*adr)[10];
  adr=new int[10][10];
  cout<<"m="; cin>>m;
  cout<<"n="; cin>>n;
  for(i=0;i<m;i++)
    for (j=0;j<n;j++)
      cin>>adr[i][j];
  for(i=0;i<m;i++)
    { for (j=0;j<n;j++) cout<<adr[i][j]<<" ";
      cout<<endl;
    }
}
```

□ **Aplicația 1.2.** Este cunoscut faptul că funcțiile nu pot întoarce masive. În schimb, o funcție poate întoarce un pointer către orice tip (**void***). Unei variabile de tipul **void*** își poate atribui orice tip de pointer, dar atribuirea inversă se poate face doar prin utilizarea operatorului de conversie explicită.

Programul următor citește două matrice și afișează suma lor. Matricele sunt alocate în **HEAP**.

```
#include <iostream.h>
void* Cit_Mat(int m, int n)
{ int i,j,(*adr1)[10]=new int[10][10];
  for(i=0;i<m;i++)
    for (j=0;j<n;j++) cin>>adr1[i][j];
  return adr1;
}

void Tip_Mat( int m,int n,int(*adr1)[10])
{ int i,j;
  for(i=0;i<m;i++)
    { for (j=0;j<n;j++) cout<<adr1[i][j]<<" ";
      cout<<endl;
    }

void* Suma_Mat( int m, int n,int(*adr1)[10],int(*adr2)[10])
{ int i,j,(*adr)[10]=new int[10][10];
  for (i=0;i<m;i++)
    for (j=0;j<n;j++)
      adr[i][j]=adr1[i][j]+adr2[i][j];
  return adr;
}
```

```
main()
{ int m,n,i,j,(*adr)[10],(*adr1)[10],(*adr2)[10];
  cout<<"m="; cin>>m;
  cout<<"n="; cin>>n;
  adr1=(int(*)[10])Cit_Mat(m,n);
  adr2=(int(*)[10])Cit_Mat(m,n);
  adr=(int(*)[10])Suma_Mat(m,n,adr1,adr2);
  Tip_Mat(m,n,adr);
}
```

Probleme propuse

1. O variabilă de tip pointer către tipul **integer/int** poate memoria:

- a) un număr întreg;
- b) conținutul unei variabile de tipul **integer/int**;
- c) adresa unei variabile de tipul **integer/int**.

2. Fie declarațiile următoare:

Varianta Pascal	Varianta C++
type adrint= ^integer ; var a1,a2: adrint ; a3: adrreal ; int *a1, *a2; float *a3;	

Care dintre atribuirile de mai jos este corectă?

- | | |
|--|--|
| a) a1:=7.35;
b) a3:=7.35;
c) a2:=a3;
d) a1:=a2; | a) a1=7.35;
b) a3=7.35;
c) a2=a3;
d) a1=a2; |
|--|--|

3. Ce înțelegeți prin **HEAP**?

- a) un segment din memorie;
- b) tipul variabilelor care rețin adrese;
- c) o variabilă de sistem în care se pot reține date de orice tip.

4. Rolul procedurii **new** / operatorului **new** este:

- a) de a crea o adresă;
- b) de a aloca spațiu pentru o variabilă la o adresă dată;
- c) de a aloca spațiu în **HEAP** pentru o variabilă de tip pointer;
- d) de a aloca spațiu în **HEAP** pentru o variabilă de un tip oarecare.

5. Rolul procedurii `dispose` / operatorului `delete` este:

- a) de a șterge conținutul unei variabile;
- b) de a șterge conținutul unei variabile alocate în **HEAP**;
- c) de a elibera spațiul ocupat de o variabilă alocate în **HEAP**;
- d) de a elibera spațiul ocupat de o variabilă oarecare;
- e) de a șterge variabila a cărei adresă se găsește memorată în **HEAP**.

6. Ce se afișează în urma executării programului următor?

Varianta Pascal	Varianta C++
<pre>type adrreal=^real; var a1,a2:adrreal; begin new(a1); a1^:=3; new (a2); a2^:=4; a2:=a1; writeln(a2^:3:0); end.</pre>	<pre>#include <iostream.h> main() { float *a1,*a2; a1=new float; *a1=3; a2=new float; *a2=4; a2=a1; cout<<*a2; }</pre>

- a) 4; b) 3; c) Eroare de sintaxă.

7. Ce se afișează în urma executării programului următor, dacă se citesc, în această ordine, valorile 7 și 8?

Varianta Pascal	Varianta C++
<pre>type adrreal=^real; var a1,a2,man:adrreal; begin new (a1); readln(a1^); new (a2); readln(a2^); man:=a2; a2:=a1; a1:=man; writeln(a1^:1:0,' ',a2^:1:0); end.</pre>	<pre>#include <iostream.h> main() { int *a1,*a2,*man; a1=new int; cin>>*a1; a2=new int; cin>>*a2; man=a2; a2=a1; a1=man; cout<<*a1<<" "<<*a2; }</pre>

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

8. Se citesc **n**, număr natural și **n** numere reale care se memorează într-un vector alocat în **HEAP**. Se cere să se afișeze media aritmetică a numerelor reținute în vector.

9. Scrieți un ansamblu de subprograme care implementează operațiile cu matrice (adunare, scădere și înmulțire). Matricele vor fi alocate în **HEAP**.

Răspunsuri

1. c); 2. d); 3. a); 4. d); 5. c); 6. b); 7. d).

Capitolul 2

Liste liniare

2.1. Definiția listelor



Definiția 2.1. O listă liniară este o colecție de $n \geq 0$ noduri, x_1, x_2, \dots, x_n aflate într-o relație de ordine. Astfel, x_1 este primul nod al listei, x_2 este al doilea nod al listei, ..., x_n este ultimul nod. Operațiile permise sunt:

- accesul la oricare nod al listei în scopul citirii sau modificării informației conținute de acesta;
- adăugarea unui nod, indiferent de poziția pe care o ocupă în listă;
- ștergerea unui nod, indiferent de poziția pe care o ocupă în listă;
- schimbarea poziției unui nod în cadrul listei.

- ⇒ Un vector poate fi privit ca o listă liniară. Oricare din operațiile de mai sus se poate efectua și pe un vector. Astfel, relația de ordine dintre elementele listei este cea a componentelor vectorului. Accesul la un nod x_i este imediat pentru că se accesează componenta i a vectorului. Adăugarea unui nod se face mai greu, pentru că nodurile care îl urmează în listă trebuie deplasate către dreapta, pentru a face loc nouului nod. Ștergerea unui nod necesită, de asemenea, efort de calcul, pentru că nodurile care urmează trebuie deplasate către stânga, pentru a ocupa spațiul lăsat liber de nodul șters. Tot așa, schimbarea poziției unui nod necesită efort de calcul. Puteti arăta, pentru această situație, în ce constă efortul de calcul?
- ⇒ Dacă o listă este memorată cu ajutorul unui vector, spunem că lista este **alocată secvențial**.



Exercițiu. Scrieți un set de subprograme cu ajutorul cărora se poate lucra cu ușurință cu o listă liniară alocată secvențial. Pentru fiecare operație permisă asupra listei veți scrie un subprogram separat.



Din căte observați, majoritatea operațiilor permise asupra unei liste liniare alocate secvențial necesită un efort mare de calcul. Din acest motiv, să simțim nevoia unei alte modalități de alocare a unei liste liniare, și anume alocarea înlănțuită¹.

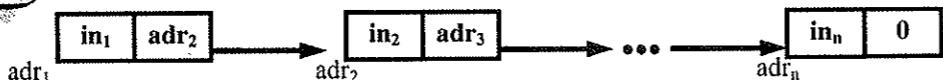
¹ A nu se face confuzie între modul de alocare a unei structuri de date, în cazul de față liste liniare și structura propriu-zisă. Aceeași structură, în exemplu lista, poate fi alocată secvențial sau înlănțuit, dar structura, conform definiției ei, rămâne aceeași.

2.2. Liste liniare alocate simplu înlăncuit

2.2.1. Prezentare generală



Definiția 2.2. O listă liniară simplu înlăncuită este o structură de formă:



Semnificația notatiilor folosite este următoarea:

- **adr₁, adr₂, adr₃, ..., adr_n** reprezintă adresele celor **n** înregistrări;
- **in₁, in₂, ..., in_n** reprezintă informațiile conținute de noduri, de altă natură decât cele de adresă;
- **0** – are semnificația "nici o adresă" – elementul este ultimul în listă.

După cum observăm, fiecare nod, cu excepția ultimului, reține adresa nodului următor.

1. Accesul la un nod al listei se face prin parcurgerea nodurilor care îl preced. Aceasta necesită un efort de calcul.
2. Informațiile de adresă sunt prezente în cadrul fiecărui nod, deci ocupă memorie.
3. Avantajele alocării înlăncuite sunt date de faptul că operațiile de adăugare sau eliminare ale unui nod se fac rapid.

Rămâne de stabilit modul în care implementăm liste liniare alocate înlăncuit. În linii mari, există două modalități: **alocarea statică și alocarea dinamică**. Pe scurt, în alocarea statică, atât datele propriu-zise asociate nodurilor (altele decât cele de adresă) cât și datele de adresă sunt memorate cu ajutorul vectorilor, iar aceștia sunt alocati în segmentul de date. În cazul **alocării dinamice**, cea pe care o prezentăm în acest capitol, toate datele sunt alocate în **Heap**².

2.2.2. Crearea și afișarea listelor

În capitolul precedent am studiat alocarea dinamică. Acum o vom aplica. Întreaga structură (lista) memorată în **HEAP** este gestionată printr-un singur pointer, memorat în segmentul de date.

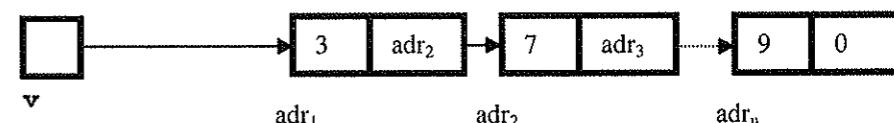
² Un exemplu de alocare statică a listelor liniare simplu înlăncuite îl veți întâlni atunci când veți studia teoria grafurilor.

- În cazul listelor, prin acel pointer se poate accesa numai primul element al listei. Apoi, pornind de la acesta se poate accesa al doilea element al listei, și.a.m.d.
- Ultimul element al listei va avea memorat în câmpul de adresă o valoare cu semnificația de nici o adresă. Cu ajutorul acestei valori programele vor detecta sfârșitul listei. În Pascal, această valoare este **nil**, iar în C++ ea este **0**.

Crearea listelor

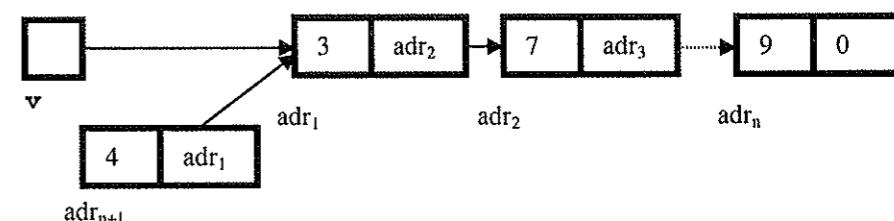
Inițial, o variabilă **v** reține **nil / 0**.

Presupunem că, la un moment dat, lista este cea de mai jos, iar **v** reține adresa primului element (**adr₁**):

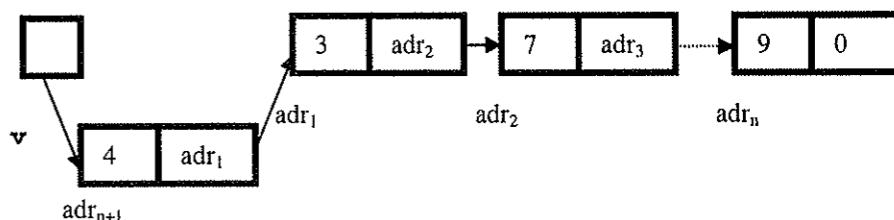


Dacă se citește un nou număr (de exemplu 4), atunci acesta se adaugă într-o înregistrare aflată la începutul listei, în următoarele etape:

- a) Se alocă spațiu pentru noua înregistrare, se completează câmpul numeric, iar adresa următoare este cea din **v**, deci a primului element al listei.



- b) Variabila **v** va memora adresa noii înregistrări:



Programul este prezentat în continuare:

Varianța Pascal	Varianța C++
<pre> type Adresa=^Nod; Nod=record info:integer; adr_urm:Adresa; end; var v:adresa; nr:integer; procedure Adaug(var v:Adresa; nr:Integer); var c:Adresa; begin new(c); c^.info:=nr; c^.adr_urm:=v; v:=c; end; procedure Tip(v:Adresa); var c:Adresa; begin c:=v; while c<>nil do begin writeln(c^.info); c:=c^.adr_urm end; begin write('numar='); readln(nr); while nr<>0 do begin Adaug(v,nr); write('numar='); readln(nr); end; Tip(v); end. </pre>	<pre> #include <iostream.h> struct Nod { int info; Nod* adr_urm; }; Nod* v; int nr; void Adaug(Nod*& v, int nr) { Nod* c=new Nod; c->info=nr; c->adr_urm=v; v=c; } void Tip(Nod* v) { Nod* c=v; while (c) { cout<<c->info<<endl; c=c->adr_urm; } } main() { cout<<"numar="; cin>>nr; while (nr) { Adaug(v,nr); cout<<"numar="; cin>>nr; }; Tip(v); } </pre>

! În Pascal, nu este permis, ca în C++, să definim un tip care conține declarări ce se referă la el. Spre exemplu, declararea de mai jos

```

Nod = record
  info:integer;
  adr_urm = ^Nod;
end;

```

este greșită. Câmpul `adr_urm` face parte din tipul `Nod` și este definit ca pointer către același tip (`Nod`). Convenția de limbaj este să se declare pe rând cele două tipuri, ca în program.

! Procedând după algoritm, lista va conține informațiile în ordinea inversă în care au fost introduse. Acest fapt nu prezintă importanță pentru majoritatea aplicațiilor.

Un alt algoritm de creare a listei, recursiv, este prezentat mai jos. De această dată lista cuprinde informațiile în ordinea în care acestea au fost introduse:

Varianța Pascal	Varianța C++
<pre> type Adresa=^Nod; Nod=record info:integer; adr_urm:Adresa; end; var v:adresa; function Adaug:Adresa; var c:adresa; nr:integer; begin write ('nr='); readln(nr); if nr<>0 then begin new(c); adaug:=c; adaug^.info:=nr; adaug^.adr_urm:=adaug end else adaug:=nil; end; procedure Tip(v:Adresa); var c:Adresa; begin c:=v; while c<>nil do begin writeln(c^.info); c:=c^.adr_urm end; begin v:=Adaug(); Tip(v); end. </pre>	<pre> #include <iostream.h> struct Nod { int info; Nod* adr_urm; }; Nod* v; Nod* Adaug() { Nod* c; int nr; cout<<"numar "; cin>>nr; if (nr) { c=new(Nod); c->adr_urm=Adaug(); c->info=nr; return c; } else return 0; } void Tip(Nod* v) { Nod*c=v; while (c) { cout<<c->info<<endl; c=c->adr_urm; } } main() { v=Adaug(); Tip(v); } </pre>

Mai jos este prezentat un subprogram recursiv care tipărește informațiile în ordine inversă față de modul în care se găsesc în listă:

Varianța Pascal	Varianța C++
<pre> procedure Tip_inv(v:adresa); begin if v<>nil then begin Tip_inv(v^.adr_urm); writeln(v^.info); end; end; </pre>	<pre> void Tip_inv(Nod* v) { if (v) { Tip_inv (v->adr_urm); cout<<v->info<<endl; } } </pre>

Problema 2.1. Fiind dată o listă liniară simplu înlățuită, cu adresa de început **v**, se cere să se inverseze legăturile din listă, adică dacă în lista inițială, după nodul **i** urmează nodul **i+1**, atunci, în noua listă, după nodul **i+1**, urmează nodul **i**.

Rezolvare. Funcția **inv**, rezolvă problema dată. Ea are doi parametri: adresa primului element al listei (**pred**) și adresa următorului element din listă (**current**). Practic, la fiecare pas, partea de adresă a nodului referit de **current** va reține adresa referită de **pred** (adică adresa nodului precedent). Funcția returnează adresa primului nod al liste inversate (adică a ultimului nod în cazul liste neinversate). Înainte de apelul funcției, partea de adresă a primului nod listei va trebui să rețină **nil/0**.

Varianța Pascal	Varianța C++
<pre>function inv(pred, current: adresa):adresa; var urm:adresa; begin while current<>nil do begin urm:=current^.adr_urm; current^.adr_urm:=pred; pred:=current; current:=urm; end; inv:=pred; end; ... current:=v^.adr_urm; v^.adr_urm:=nil; v:=inv(v,current); Tip(v); end;</pre>	<pre>Nod* inv(Nod* pred, Nod* current) { Nod* urm; while (current) { urm=current->adr_urm; current->adr_urm=pred; pred=current; current=urm; } return pred; } Nod* current=v->adr_urm; v->adr_urm=0; v=inv(v,current); Tip(v);</pre>

 **Exercițiu.** Modificați subprogramul astfel încât acesta să aibă un singur parametru, adresa de început a listei.

2.2.3. Operații asupra unei liste liniare

În acest paragraf prezentăm principalele operații care se pot efectua cu o listă liniară simplu înlățuită.

 În prezentarea operațiilor, vom folosi de multe ori desene. Rețineți: pentru orice operație aveți de efectuat, faceți un mic desen. Vă ajută mult...

Orice listă va fi reținută prin două informații de adresă: a primului nod (**v**) și a ultimului nod (**sf**). Precizăm faptul că, în general, numai prima informație este indispensabilă. Pentru simplitate și pentru rapiditatea executării vom reține și adresa ultimului nod. Structura unui nod al listei este:

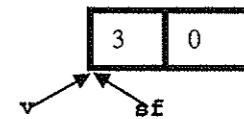
Varianța Pascal	Varianța C++
<pre>type Adresa=^Nod; Nod=record info:integer; adr_urm:Adresa; end;</pre>	<pre>struct Nod { int info; Nod* adr_urm; };</pre>

A. Adăugarea unui nod

Fiind dată o listă liniară, se cere să se adauge la sfârșitul ei un nod, cu o anumită informație, în exemplele noastre, un număr întreg. Se disting două cazuri:

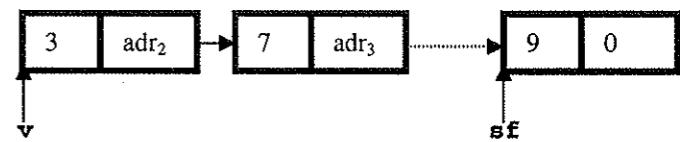
a) lista este vidă - **v** reține 0.

Să presupunem că vrem să adăugăm un nod cu informația 3. Se alocă în **HEAP** nodul respectiv, adresa sa va fi în **v**, și cum lista are un singur nod, adresa primului nod este și adresa ultimului, deci conținutul lui **v** va coincide cu acela al lui **sf**.

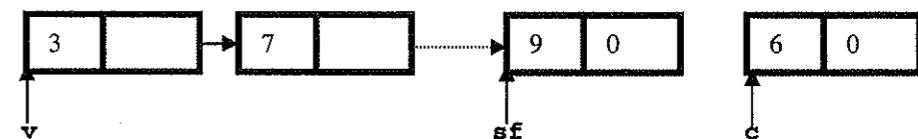


b) lista este nevidă

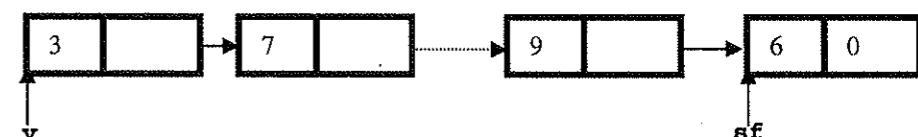
Fie lista:



Se adaugă un nod cu informația 6. Inițial se alocă spațiu pentru nod.



Câmpul de adresă al ultimului nod, cel care are adresa în **sf**, va reține adresa nodului nou creat, după care și **sf** va reține aceeași valoare.



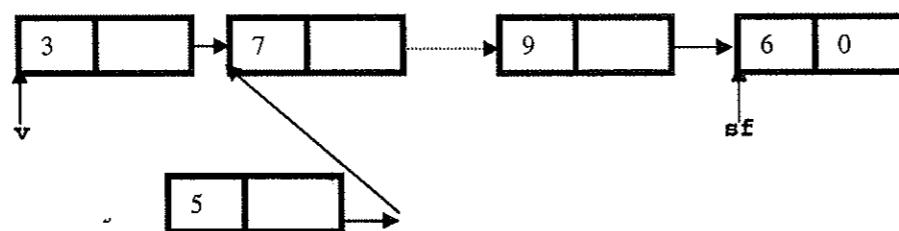
! Dacă n-am fi utilizat variabila `sf` pentru a reține adresa ultimului nod, ar fi fost necesar să parcurgem întreaga listă, pornind de la `v`, pentru a obține adresa ultimului.

Varianta Pascal	Varianta C++
<pre>procedure Adaugare(var v, sf:Adresa;val:integer); var c:Adresa; begin if v=nil then begin new(v); v^.info:=val; v^.adr_urm:=nil; sf:=v; end else begin New(c); sf^.adr_urm:=c; c^.info:=val; c^.adr_urm:=nil; sf:=c; end end;</pre>	<pre>void Adaugare(Nod*& v, Nod*& sf, int val) { Nod* c; if (v==0) { v=new(Nod); v->info=val; v->adr_urm=0; sf=v; } else { c=new(Nod); sf->adr_urm=c; c->info=val; c->adr_urm=0; sf=c; } }</pre>

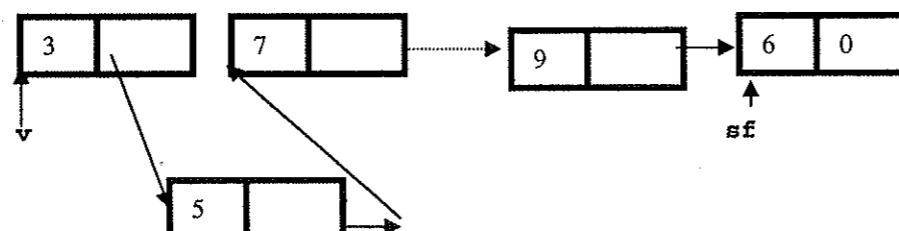
B. Inserarea unui nod, după un altul, de informație dată

Fie lista din figura anterioară. Dorim să adăugăm după nodul cu informația 3, un altul, cu informația 5.

Inițial, se identifică nodul după care se face adăugarea. În cazul de față acesta este primul. Se alocă spațiu pentru noul nod. Se completează adresa și numele adresa nodului care urmează după cel de informație 3.



Apoi, câmpul de adresă al nodului cu informația 3 va reține adresa nodului nou creat:



! Un caz aparte apare atunci când nodul de informație `val` este ultimul în listă. În acest caz `sf` va reține adresa nodului nou creat pentru că acesta va fi ultimul.

Varianta Pascal	Varianta C++
<pre>procedure Inserare_dupa(v:adresa;var sf:adresa; val,val1:integer); var c,d:adresa; begin c:=v; while c->info<>val do c:=c->adr_urm; new(d); d^.info:=val1; d^.adr_urm:=c^.adr_urm; c^.adr_urm:=d; if d^.adr_urm=nil then sf:=d; end;</pre>	<pre>void Inserare_dupa(Nod* v, Nod*& sf, int val, int val1) { Nod* c=v, *d; while (c->info!=val) c=c->adr_urm; d=new Nod; d->info=val1; d->adr_urm=c->adr_urm; c->adr_urm=d; if (d->adr_urm==0) sf=d; }</pre>

C. Inserarea unui nod, înaintea altuia, de informație dată

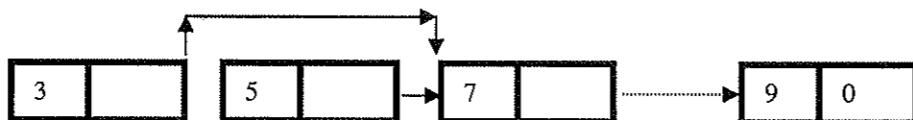
Întrucât operația este asemănătoare cu precedenta, prezentăm numai subprogramul care realizează operația respectivă:

Varianta Pascal	Varianta C++
<pre>procedure Inserare_inainte(var v:adresa;val,val1:integer); var c,d:adresa; begin if v^.info==val then begin new(d); d^.info:=val1; d^.adr_urm:=v; v=d; end else begin c:=v; while (c->adr_urm^.info<> val) c:=c->adr_urm; new(d); d^.info:=val1; d^.adr_urm:=c^.adr_urm; c^.adr_urm:=d; end end;</pre>	<pre>void Inserare_inainte(Nod*& v, int val, int val1) { Nod* c,*d; if (v->info==val) { d=new Nod; d->info=val1; d->adr_urm=v; v=d; } else { c=v; while (c->adr_urm-> info!=val) c=c->adr_urm; d=new Nod; d->info=val1; d->adr_urm=c->adr_urm; c->adr_urm=d; } }</pre>

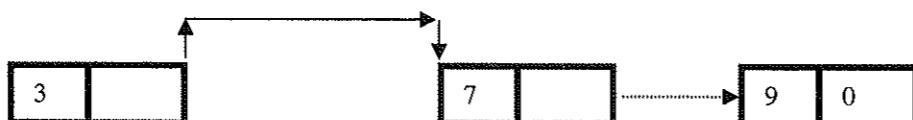
D. Ștergerea unui nod de informație dată

Algoritmul este diferit în funcție de poziția în listă a nodului care va fi șters - dacă este primul sau nu.

a) **Nodul nu este primul.** Pentru nodul care va fi șters, informația de adresă a predecesorului va reține adresa nodului succesor:



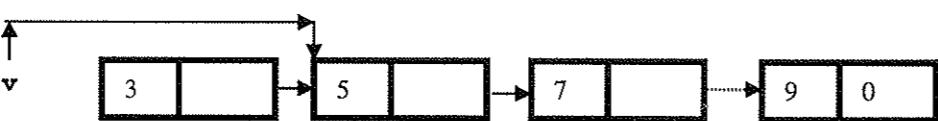
Memoria ocupată de nodul care urmează să fie șters este eliberată:



b) **Nodul este primul.** Fie lista:



Variabila **v** va reține adresa celui de-al doilea nod:



Spațiul ocupat de primul nod va fi eliberat:



Programul este următorul:

Varianta Pascal	Varianta C++
<pre>procedure Sterg(var v, sf:adresa;val:integer); var c,man:adresa; begin if v^.info=val then begin man:=v; v:=v^.adr_urm; end</pre>	<pre>void Sterg(Nod*& v, Nod*& sf, int val) { Nod* c, *man; if (v->info==val) { man=v; v=v->adr_urm; }</pre>

<pre>else begin c:=v; while c^.adr_urm^.info<>val do c:=c^.adr_urm; man:=c^.adr_urm; c^.adr_urm:=man^.adr_urm; if man=sf then sf:=c; end; dispose(man); end;</pre>	<pre>else { c=v; while (c->adr_urm->info !=val) c=c->adr_urm; man=c->adr_urm; c->adr_urm=man->adr_urm; if (man==sf) sf=c; } delete man;</pre>
--	---

Pentru a verifica modul de funcționare a subprogramelor de mai sus, este necesar să utilizăm un altul, care afișează lista liniară:

Varianta Pascal	Varianta C++
<pre>procedure listare(v:Adresa); var c:Adresa; begin c:=v; while c<>nil do begin write(c^.info, ' '); c:=c^.adr_urm; end; writeln; end;</pre>	<pre>void Listare(Nod* v) { Nod* c=v; while (c) { cout<<c->info<<endl; c=c->adr_urm; } cout<<endl; }</pre>

Acum putem testa aplicația. Utilizați secvența următoare:

Varianta Pascal	Varianta C++
<pre>var v,sf:Adresa; i:integer; begin for i:=1 to 10 do Adaugare(v,sf,i); listare(v); inserare_dupa(v,sf,7,11); inserare_dupa(v,sf,10,12); inserare_dupa(v,sf,1,13); listare(v); inserare_inainte(v,13,14); inserare_inainte(v,1,15); listare(v); sterg(v,sf,15); sterg(v,sf,13); sterg(v,sf,12); listare(v); end.</pre>	<pre>Nod* v,*sf; int i; main() { for (i=1;i<=10;i++) Adaugare(v,sf,i); Listare(v); Inserare_dupa(v,sf,7,11); Inserare_dupa(v,sf,10,12); Inserare_dupa(v,sf,1,13); Listare(v); Inserare_inainte(v,13,14); Inserare_inainte(v,1,15); Listare(v); Sterg(v,sf,15); Sterg(v,sf,13); Sterg(v,sf,12); Listare(v); }</pre>

2.2.4. Aplicații ale listelor liniare

2.2.4.1. Sortarea prin inserție

Se citesc de la tastatură n numere naturale. Se cere ca acestea să fie sortate crescător prin utilizarea metodei de sortare prin inserție.

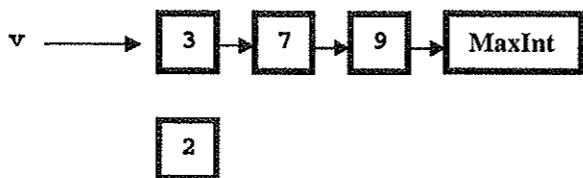
Această metodă de sortare a fost studiată în clasa a IX-a. Ideea de bază a metodei constă în a considera primele k valori sortate, urmând să inserăm valoarea $k+1$ în sirul deja sortat. Prin utilizarea listelor liniare înlățuite, inserția este mai simplă, întrucât nu necesită deplasarea componentelor, ca în cazul vectorilor.

Pentru simplificarea algoritmului, lista va conține valoarea maximă **MaxInt** alocată deja în listă. În acest fel, algoritmul se simplifică pentru că se pornește de la o listă liniară nevidă. Evident, valoarea **MaxInt** nu va fi listată, atunci când se tipăresc numerele sortate.

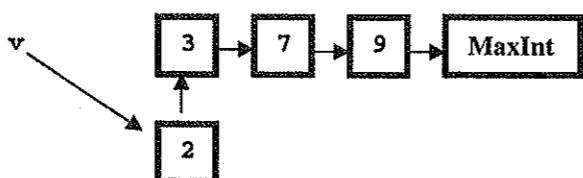
Problema se reduce la inserția unui număr într-o listă deja sortată. Mai întâi se alocă spațiu în **HEAP** pentru o valoare, apoi aceasta este citită. Se disting două cazuri:

1. Valoarea citită este mai mică decât prima valoare a listei. Aceasta înseamnă că ea este cea mai mică din listă și va fi introdusă prima în listă.

Ex: Fie lista următoare și se citește 2:



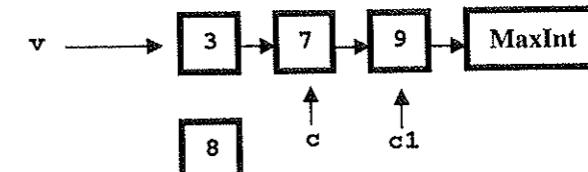
Noua înregistrare va conține adresa nodului 3, iar **v** conține adresa noii înregistrări:



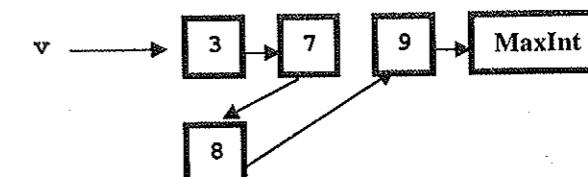
2. Valoarea citită nu este cea mai mică din listă. În mod sigur, nu este cea mai mare, pentru că am introdus **MaxInt** în listă. Dacă nu a fost îndeplinită condiția de la cazul 1, înseamnă că valoarea nu este nici cea mai mică. Aceasta înseamnă că ea va trebui introdusă în interiorul listei.

Va trebui să identificăm prima valoare mai mare decât valoarea citită. Întrucât, odată găsită adresa acestei valori, avem nevoie de adresa precedentă (pentru a putea lega în listă noul nod) vom "merge" cu doi pointeri, **c** și **c1**, unde **c1** reține adresa înregistrării cu valoare mai mare decât înregistrarea citită, iar **c** adresa înregistrării precedente.

Ex: Fie lista următoare și se citește 8. Se identifică prima înregistrare care reține o valoare mai mare decât cea citită (în exemplu, 9).



Noua valoare se inserează în listă.



Varianta Pascal	Varianta C++
<pre> type Adresa=^Nod; Nod=record info:integer; adr_urm:Adresa; end; var n,i:integer; var v,adr,c,c1:Adresa; begin write('n='); readln(n); new(v); v^.info:=MaxInt; v^.adr_urm:=nil; for i:=1 to n do begin new(adr); write('numar='); readln(adr^.info); if adr^.info<=v^.info then (primul din lista) begin adr^.adr_urm:=v; v:=adr; end end end. </pre>	<pre> #include <iostream.h> const MaxInt=32000; struct Nod { int info; Nod* adr_urm; }; int n,i; Nod *v,*adr,*c,*c1; main() { cout<<"n="; cin>>n; v=new Nod; v->info=MaxInt; v->adr_urm=0; for (i=1;i<=n;i++) { adr=new Nod; cout<<"numar="; cin>>adr->info; if (adr->info<=v->info) // primul din lista { adr->adr_urm=v; v=adr; } } } </pre>

```

else{nu e primul din lista)
begin
  c:=v;
  c1:=v^.adr_urm;
  while c1^.info<adr^.info
    do
      begin
        c:=c^.adr_urm;
        c1:=c1^.adr_urm;
      end;
      c^.adr_urm:=adr;
      adr^.adr_urm:=c1;
    end;
end;
{tiparesc}
c:=v;
while c^.info<>MaxInt do
begin
  writeln(c^.info);
  c:=c^.adr_urm
end
end.

```

```

else
  // nu e primul din lista
  { c=v;
    c1=v^.adr_urm;
    while (c1->info<adr->info)
      { c=c->adr_urm;
        c1=c1->adr_urm;
      }
    c->adr_urm=adr;
    adr->adr_urm=c1;
  }
  //tiparesc
  c=v;
  while (c->info!=MaxInt)
  { cout<<c->info<<endl;
    c=c->adr_urm;
  }

```

! Procedeul generării unei valori mai mari sau mai mici decât toate cele posibile este deseori folosit în programare. Realizați cât de mult a simplificat algoritmul?

La fiecare adăugare în listă se face o parcurgere a acesteia, deci algoritmul are complexitatea maximă $O(n^2)$.

2.2.4.2. Sortarea topologică

Presupunem că dorim sortarea numerelor $1, 2, \dots, n$, numere care se găsesc într-o ordine oarecare, alta decât cea naturală. Pentru a afla relația în care se găsesc numerele, introducem un număr finit de perechi (i, j) . O astfel de pereche ne exprimă faptul că, în relația de ordine considerată, i se află înaintea lui j .

Ex: Fie $n=3$ și citim perechile $(3, 1)$ și $(3, 2)$. Numărul 3 se află înaintea lui 1 și 3 se află înaintea lui 2. Apar două soluții posibile: $3, 1, 2$ și $3, 2, 1$, întrucât nu avem nici o informație asupra relației dintre 1 și 2.

De aici tragem concluzia că o astfel de problemă poate avea mai multe soluții.

Ex: Fie $n=3$ și citim $(1, 2)$, $(2, 3)$, $(3, 1)$. În acest caz nu avem soluție. Din primele două relații rezultă că ordinea ar fi $1, 2, 3$, iar relația a-3-a contrazice această ordine.

În concluzie, problema poate avea sau nu soluție, iar dacă are, poate fi unică sau nu.

Algoritmul pe care îl prezentăm în continuare furnizează o singură soluție atunci când problema admite soluții. În caz contrar, specifică faptul că problema nu admite soluție.

Vom exemplifica funcționarea algoritmului pentru $n=4$ și citind perechile $(3, 4)$, $(4, 2)$, $(1, 2)$, $(3, 1)$.

Pentru fiecare număr între 1 și n trebuie să avem următoarele informații:

- numărul predecesorilor;
- lista succesorilor.

Pentru aceasta folosim doi vectori:

- **contor**, vector care reține numărul predecesorilor fiecărui k , $k \in \{1 \dots n\}$;
- **a**, care reține adresele de început ale listelor de succesiuni ai fiecărui element.

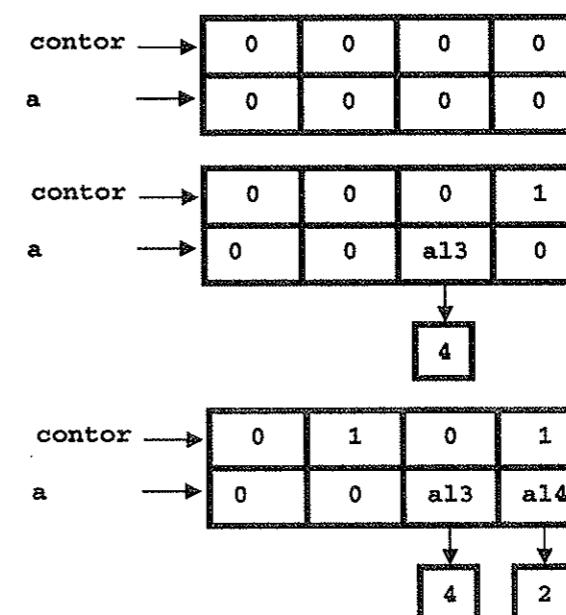
Pentru fiecare element există o listă simplu înălțuită a succesorilor săi.

Inițial, în dreptul fiecărui element din vectorii **contor** și **a** se trece 0.

Citirea unei perechi (i, j) înseamnă efectuarea următoarelor operații:

- incrementarea variabilei **contor(j)** (j are un predecesor, și anume i);
- adăugarea lui j la lista succesorilor lui i .

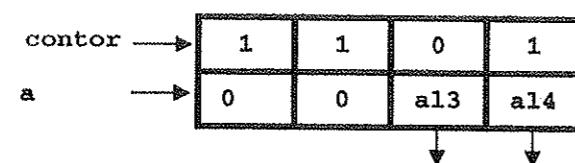
Pentru exemplul dat, se procedează în felul următor:



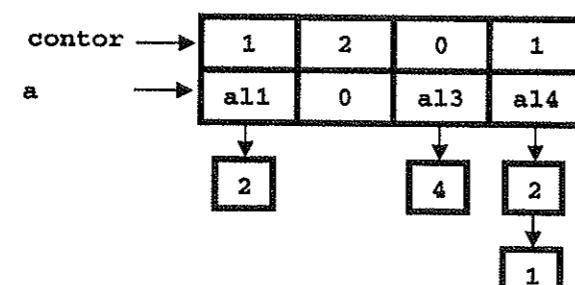
Valorile inițiale ale celor doi vectori.

am citit $(3, 4)$;
a13 - adresa listei 3;

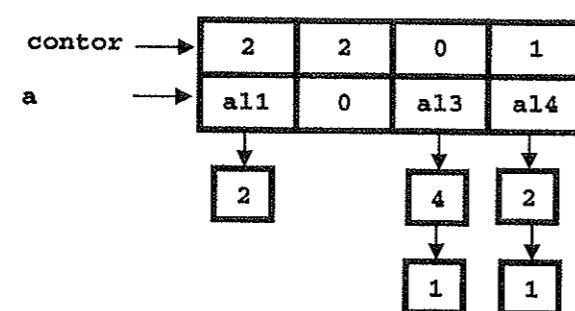
am citit $(4, 2)$;



am citit (4,1);



am citit (1,2);

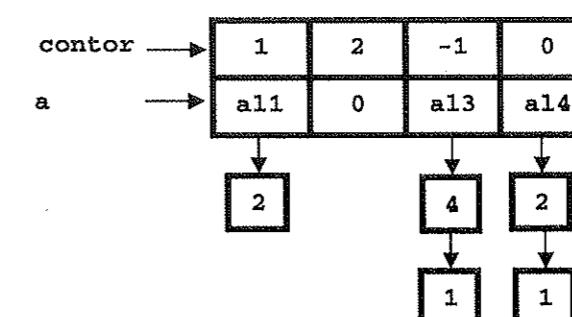


am citit (3,1);

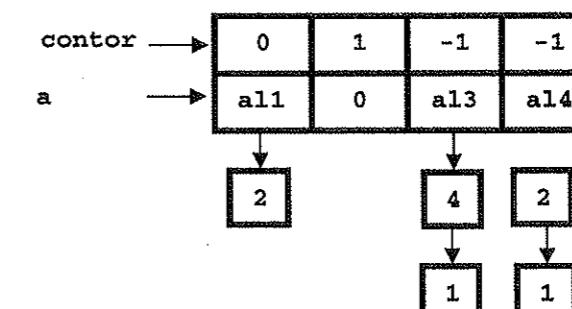
În continuare se procedează astfel:

- ⇒ toate elementele care au 0 în câmpul **contor** se rețin într-un vector **c**;
- ⇒ pentru fiecare element al vectorului **c** se procedează astfel:
 - se tipărește;
 - se marchează cu -1 câmpul său de contor;
 - pentru toți succesorii săi (aflați în lista succesorilor) se scade 1 din câmpul **contor** (este normal, încrucișat aceștia au un predecesor mai puțin);
- ⇒ se reia algoritmul dacă nu este îndeplinită una din condițiile următoare:
 - au fost tipărite toate elementele, caz în care algoritmul se încheie cu succes;
 - nu avem nici un element cu 0 în câmpul **contor**, caz în care relațiile au fost incoerente.

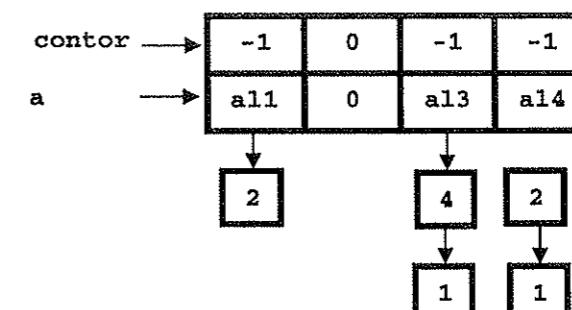
Astfel:



Tipăresc 3, scad 1 din predecesorii lui 4 și 1, marchez cu -1 contorul lui 3.



Tipăresc 4, scad 1 din predecesorii lui 2 și 1, marchez cu -1 contorul lui 4.



Tipăresc 1, scad 1 din predecesorii lui 2, marchez cu -1 contorul lui 1.

Algoritmul are multe aplicații, ca de exemplu:

- ordonarea unor activități, atunci când ele sunt condiționate una de alta;
- ordonarea unor termeni care se cer explicații, pentru a-i putea explica prin alții deja prezentați.

Algoritmul are complexitatea $O(n^2)$. Odată sunt **n** extrageri și la fiecare extragere se parurge lista succesorilor unui nod.

Varianta Pascal	Varianta C++
<pre> type ref=^inr; inr=record succ:integer; urm:ref; end; vector=array [1..100] of integer; vectad=array [1..100] of ref; var n,m,i,j,k:integer; contor,c:vector; a:vectad; gasit:boolean; procedure adaug(i,j:integer); var c,d:ref; begin contor[j]:=contor[j]+1; c:=a[i]; new(d); d^.urm:=nil; d^.succ:=j; if c=nil then a[i]:=d else begin while c^.urm<>nil do c:=c^.urm; c^.urm:=d end; end; procedure actual(i:integer); var c:ref; begin c:=a[i]; while c<>nil do begin contor[c^.succ]:=contor[c^.succ]-1; c:=c^.urm end; begin write('n='); readln(n); for i:=1 to n do begin contor[i]:=0; a[i]:=nil end; end; end; </pre>	<pre> #include <iostream.h> struct Nod { int succ; Nod* urm; }; int n,m,i,j,k,gasit; int contor[100],c[100]; Nod* a[100]; void adaug(int i,int j) { Nod *c, *d; contor[j]++; c=a[i]; d=new Nod; d->urm=0; d->succ=j; if (c==0) a[i]=d; else { while (c->urm) c=c->urm; c->urm=d; } } void actual(int i) { Nod* c=a[i]; while (c) { contor[c->succ]--; c=c->urm; } } main() { cout<<"n="; cin>>n; for (i=1;i<=n;i++) { contor[i]=0; a[i]=0; } while (i) { cout<<"Tastati i, j="; cin>>i>>j; if (i) adaug(i,j); } m=n; do { k=1; gasit=0; for (i=1;i<=n;i++) if (contor[i]==0) { gasit=1; m--; c[k]=i; k++; contor[i]=-1; } for (i=1;i<=k-1;i++) { actual(c[i]); cout<<c[i]<<endl; } } } </pre>

```

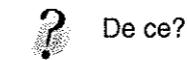
while i<>0 do
begin
  write('Tastati i,j=');
  readln(i,j);
  if i<>0 then adaug(i,j)
end;
m:=n;
repeat
k:=1;
gasit:=false;
for i:=1 to n do
  if contor[i]=0
  then
    begin
      gasit:=true;
      m:=m-1;
      c[k]:=i;
      k:=k+1;
      contor[i]:=-1
    end;
  for i:=1 to k-1 do
    begin
      actual(c[i]);
      writeln(c[i]);
    end;
until (not gasit) or (m=0);
if m=0
  then writeln('totul e ok')
  else writeln('relatii
contradictorii')
end.

```

2.2.4.3. Operații cu polinoame

În acest paragraf vom prezenta modul în care se pot programa operații precum adunarea, scăderea, înmulțirea și împărțirea polinoamelor cu coeficienți reali.

Să observăm că, în acest caz, utilizarea listelor liniare simplu înlăncuite este necesară și, exemplul în sine, constituie un argument pentru utilizarea acestora.



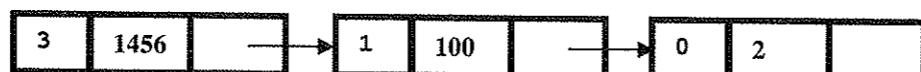
De ce?

Să presupunem că un polinom va fi memorat cu ajutorul unui vector. Fiecare coeficient va fi memorat de o componentă a vectorului. De exemplu, polinomul: $10x^3 + 6x^2 + 3x + 2$, poate fi memorat cu ajutorul vectorului (10,6,3,2). Dar dacă avem polinomul $3x^{1456} + 1$, cum procedăm? Este necesar să avem un vector cu 1457 de componente, dintre care numai prima și ultima sunt diferite de 0. Este eficient să folosim vectori? Evident, nu.



Atunci?

- Pentru memorarea unui polinom (prin coeficienții lui) vom utiliza o listă liniară simplu înlățuită. Fiecare nod al listei va reține, în această ordine, coeficientul și gradul unui monom. Pentru simplificarea operațiilor cu polinoame, un polinom va fi reținut în ordinea descrescătoare a gradelor. De exemplu, pentru polinomul $3x^{1456} + x^{100} + 2$, vom avea:



Mai jos, prezentăm structura unui nod al listei:

Varianta Pascal	Varianta C++
<pre> type adrNod^=Nod; Nod=record coef:real; grad:integer; adr_urm:adrNod; end; </pre>	<pre> struct Nod { float coef; int grad; Nod* adr_urm; }; </pre>

1. Pentru a adăuga un nod listei liniare simplu înlățuite, atunci când aceasta se creează, vom utiliza subprogramul următor, **adaug**. Parametrii de intrare sunt adresele de început și de sfârșit ale listei. Desigur, se poate evita parametrul prin care se transmite adresa de sfârșit a listei, dar, prin transmiterea acestui parametru, adăugarea unui nod la sfârșitul listei se poate face cu mult mai repede, pentru că nu mai este necesară parcurgerea întregii liste.

Varianta Pascal	Varianta C++
<pre> procedure adaug(var v, sf:adrNod; gr:integer; cf:real); var c:adrNod; begin new(c); c^.grad:=gr; c^.coef:=cf; c^.adr_urm:=nil; if v=nil then begin v:=c; sf:=c; end else begin sf^.adr_urm:=c; sf:=c; end end; </pre>	<pre> void adaug(Nod*& v, Nod*& sf, int gr, float cf) { Nod* c; c=new Nod; c->grad=gr; c->coef=cf; c->adr_urm=0; if (v==0) v=sf=c; else { sf->adr_urm=c; sf=c; } } </pre>

2. Pentru a crea lista asociată unui polinom, vom utiliza funcția următoare, care după ce creează lista, returnează adresa ei de început. Pentru a putea introduce datele, se cere, de la început, numărul de "termeni" (monoame) ai polinomului. Este foarte important ca monoamele să fie introduse *în ordinea descrescătoare a gradelor*, pentru că funcția nu realizează ordonarea acestora. Pentru a crea lista se utilizează subprogramul prezentat anterior, **adaug**. **Exercițiu.** Modificați funcția astfel încât datele de intrare să se găsească într-un fișier text. De asemenea, se cere ca funcția să sorteze monoamele *în ordinea descrescătoare a gradului*.

Varianta Pascal	Varianta C++
<pre> function crePolinom(nr_termeni: integer):adrNod; var gradul,i:integer; coeficient:real; vf,sf:adrNod; begin vf:=nil; for i:=1 to nr_termeni do begin write ('Grad='); readln(gradul); write ('coeficientul='); readln(coeficient); adaug(vf,sf,gradul, coeficient); end; crePolinom:=vf; end; </pre>	<pre> Nod* crePolinom(int nr_termeni) { cout<<"Date polinom "<<endl; Nod* vf=0, *sf; int gradul; float coeficient; for (int i=1;i<=nr_termeni;i++) { cout<<"Grad="; cin>>gradul; cout<<"coeficientul="; cin>>coeficient; adaug(vf,sf,gradul, coeficient); } return vf; } </pre>

3. Uneori, în operațiile care vor fi prezentate, intervin anumite polinoame, de care este nevoie numai la un moment dat, după care devin inutile. Fiind polinoame, sunt memorate tot sub formă unor liste liniare simplu înlățuite. Care este problema? După utilizarea lor, aceste liste trebuie șterse, pentru că ocupă în mod inutil memoria. Pentru a șterge o listă, utilizăm subprogramul **sterg**. Să observăm că ștergerea listei se face, prin parcurgerea ei, nod cu nod. Imediat ce un nod a fost parcurs, el este șters.

Varianta Pascal	Varianta C++
<pre> procedure sterg(v:adrNod); var c:adrNod; begin c:=v; while v<>nil do begin v:=v^.adr_urm; dispose(c); c=v; end end; </pre>	<pre> void sterg(Nod* v) { Nod * c=v; while (v) { v=v->adr_urm; delete c; c=v; } } </pre>

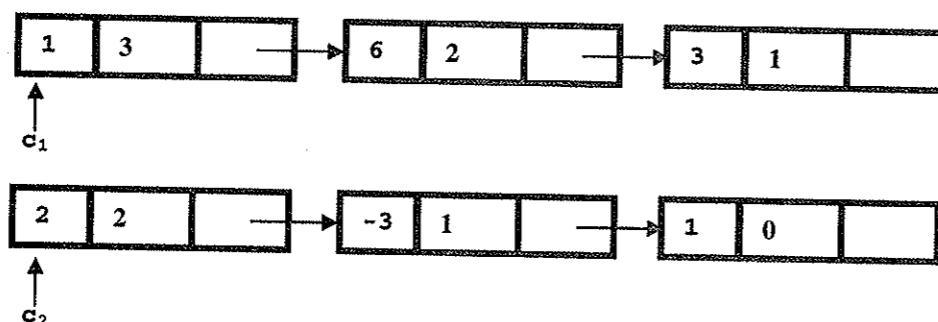
4. După ce efectuăm o operație cu polinoame, este necesar ca rezultatul să fie afișat. Pentru afișarea unui polinom vom utiliza subprogramul următor:

Varianta Pascal	Varianta C++
<pre>procedure afis(v:adrNod); var i:integer; begin i:=0; while v<>nil do begin if i<>0 then write('+'); write(v^.coef:3:2,'x**', v^.grad); v:=v^.adr_urm; i++; end; writeln; writeln; end;</pre>	<pre>void afis(Nod* v) { int i=0; while (v) { if (i) cout<<"+"<< v->coef<<"x**"<< v->grad; v=v->adr_urm; i++; } cout<<endl<<endl; }</pre>

5. **Adunarea polinoamelor.** Pentru a aduna două polinoame se utilizează funcția **adun**. Cele două polinoame care se adună sunt sub forma unor liste liniare simplu înălțuită. Nu uitați, nodurile listelor conțin monoame, iar pentru fiecare monom se cunoaște gradul și coeficientul său. Monoamele sunt aranjate în ordinea descrescătoare a gradului. Funcția returnează adresa de început a polinomului sumă. Să presupunem că avem de adunat polinoamele:

$$x^3+6x^2+3x \text{ și } 2x^2-3x+1.$$

Listele asociate celor două polinoame sunt prezentate mai jos:



Algoritmul de adunare a polinoamelor seamănă foarte mult cu algoritmul de interclasare a două șiruri ordonate. Avem două variabile de tip pointer, **c₁** și **c₂**. Variabila **c₁** reține adresa de început a primei liste, iar **c₂** adresa de început a celeilalte liste.

Aveam mai multe situații:

- a) dacă gradul monomului indicat de **c₁** este egal cu gradul monomului indicat de **c₂** și dacă suma coeficienților celor două monoame este diferită de 0, atunci se adaugă la lista polinomului rezultat monomul care are gradul comun și care are drept coeficient suma celor doi coeficienți. Indiferent de suma coeficienților, avansează **c₁** și **c₂**;
- b) dacă gradul monomului indicat de **c₁** este strict mai mare decât gradul monomului indicat de **c₂**, se adaugă la lista polinomului rezultat monomul indicat de **c₁**, avansează **c₁**;
- c) dacă gradul monomului indicat de **c₁** este strict mai mic decât gradul monomului indicat de **c₂**, se adaugă la lista polinomului rezultat monomul indicat de **c₂**, avansează **c₂**.

Testele de mai sus, se repetă cât timp nu s-a parcurs integral nici o listă de intrare. Dacă s-a ajuns la sfârșitul listei pointată de **c₁**, atunci lista pointată de **c₂** este copiată, începând de la **c₂** până la sfârșit în lista rezultat. Dacă s-a ajuns la sfârșitul listei pointată de **c₂**, atunci lista pointată de **c₁** este copiată, începând de la **c₁** până la sfârșit în lista rezultat.

Pentru x^3+6x^2+3x și $2x^2-3x+1$, **c₁** pointează către nodul asociat monomului x^3 , iar **c₂** pointează către nodul asociat monomului $2x^2$.

- Gradul monomului reținut de nodul pointat de **c₁** este mai mare decât gradul monomului reținut de nodul pointat de **c₂**. Primul nod al listei polinomului rezultat va fi x^3 . **c₁** va pointa către nodul care reține monomul $6x^2$.
- Gradele celor două monoame sunt egale și suma coeficienților este diferită de 0. Al doilea nod al listei polinomului rezultat va fi $8x^2$. **c₁** va pointa către nodul care reține monomul $3x$, iar **c₂** va pointa către nodul care reține monomul $-3x$.
- Gradele celor două monoame sunt egale și suma coeficienților este egală cu 0. Prin avansul lui **c₁**, prima listă a fost parcursă integral.
- Al treilea nod al listei polinomului rezultat va reține 1.

Varianta Pascal	Varianta C++
<pre>function adun(v1, v2:adrNod):adrNod; var c1,c2,v,sf:adrNod; begin c1:=v1; c2:=v2; v:=nil;</pre>	<pre>Nod* adun(Nod* v1,Nod* v2) { Nod* c1=v1; Nod* c2=v2; Nod* v=0, *sf;</pre>

```

while (c1<>nil) and (c2<>nil)
do
  if c1^.grad=c2^.grad
  then
    begin
      if c1^.coef+c2^.coef<>0
      then
        adaug(v,sf,c1^.grad,
               c1^.coef+c2^.coef);
      c1:=c1^.adr_urm;
      c2:=c2^.adr_urm;
    end
  else
    if c1^.grad>c2^.grad
    then
      begin
        adaug(v,sf,c1^.grad,
               c1^.coef);
        c1:=c1^.adr_urm;
      end
    else
      begin
        adaug(v,sf,c2^.grad,
               c2^.coef);
        c2:=c2^.adr_urm;
      end;
  if c1<>nil
  then
    while c1<>nil do
      begin
        adaug(v,sf,c1^.grad,
               c1^.coef);
        c1:=c1^.adr_urm;
      end
  else
    while c2<>nil do
      begin
        adaug(v,sf,c2^.grad,
               c2^.coef);
        c2:=c2^.adr_urm;
      end;
  adun:=v;
end;

```

6. Scăderea polinoamelor. Pentru această operație vom folosi o funcție **negativ**, care primește ca parametru de intrare adresa de început a unei liste liniare care reține un polinom și returnează adresa de început a unei noi liste care reține un polinom în care coeficienții sunt cei ai polinomului de intrare înmulțiti cu -1.

Evident, scăderea se reduce la aplicarea funcției **negativ** pentru polinomul descăzut și adunarea rezultatului cu scăzătorul.

```

while (c1 && c2)
  if (c1->grad==c2->grad)
  { if (c1->coef+c2->coef)
    adaug(v,sf,c1->grad,
           c1->coef+c2->coef);
    c1=c1->adr_urm;
    c2=c2->adr_urm;
  }
  else
    if (c1->grad>c2->grad)
    { adaug(v,sf,c1->grad,
              c1->coef);
      c1=c1->adr_urm;
    }
    else
      { adaug(v,sf,c2->grad,
                c2->coef);
        c2=c2->adr_urm;
      }
  if (c1)
    while (c1)
    { adaug(v,sf,c1->grad,
              c1->coef);
      c1=c1->adr_urm;
    }
  else
    while (c2)
    { adaug(v,sf,c2->grad,
              c2->coef);
      c2=c2->adr_urm;
    }
  return v;
}

```

Varianta Pascal	Varianta C++
<pre> function negativ(v:adrNod) :adrNod; var v1,sf1:adrNod; begin v1:=nil; while v<> nil do begin adaug(v1,sf1,v^.grad, -v^.coef); v:=v^.adr_urm; end; negativ:=v1; end; </pre>	<pre> Nod* negativ(Nod* v) { Nod *v1=0, *sf1; while (v) { adaug(v1,sf1,v->grad, -(v->coef)); v=v->adr_urm; } return v1; } </pre>

7. Înmulțirea unui polinom cu un monom. Pentru a realiza această operație utilizăm funcția **mulMonom**. Funcția primește ca parametri de intrare adresa de început a listei care reține un polinom, gradul monomului și coeficientul său. Funcția construiește un nou polinom, cel rezultat în urma înmulțirii polinomului dat cu monomul și returnează adresa de început a listei care îl reține.

Varianta Pascal	Varianta C++
<pre> function mulMonom(v:adrNod; gr:integer;cf:real):adrNod; var vf,sf:adrNod; begin vf:=nil; while v<> nil do begin adaug(vf,sf,v^.grad+gr, v^.coef*cf); v:=v->adr_urm; end; mulMonom:=vf; end; </pre>	<pre> Nod* mulMonom(Nod* v,int gr, float cf) { Nod* vf=0, *sf; while (v) { adaug(vf,sf,v->grad+gr, v->coef*cf); v=v->adr_urm; } return vf; } </pre>

8. Înmulțirea a două polinoame. Operația este realizată de funcția **mul**. Ea are ca parametri de intrare adresele de început ale listelor care rețin polinomul deînmulțit și polinomul înmulțitor și returnează o nouă listă în care se găsește polinomul rezultat.

Produsul se obține înmulțind, cu fiecare monom al polinomului înmulțitor, polinomul deînmulțit și adunarea fiecărui polinom astfel obținut la rezultat.

Observați cum se șterg polinoamele intermediare, care nu mai sunt necesare!

Varianta Pascal	Varianta C++
<pre>function mul(v1,v2:adrNod):adrNod; var v,vman,vman1:adrNod; begin v:=nil; while v2<>nil do begin vman:=mulMonom(v1,v2^.grad, v2^.coef); vman1:=v; v:=adun(v,vman); sterg(vman); sterg(vman1); v2:=v2^.adr_urm; end; mul:=v; end;</pre>	<pre>Nod* mul(Nod* v1, Nod* v2) { Nod* v=0; while (v2) { Nod *vman=0, *vman1; vman=mulMonom(v1, v2->grad,v2->coef); vman1=v; v=adun(v,vman); sterg(vman); sterg(vman1); v2=v2->adr_urm; } return v; }</pre>

9. Împărțirea polinoamelor. Operația este realizată de subprogramul `divp`. Acesta primește ca parametri de intrare adresele de început ale listelor care rețin polinomul deîmpărțit și polinomul împărțitor. Subprogramul returnează adresele de început ale listelor care rețin polinomul cât și polinomul rest. Inițial, restul va fi deîmpărțitul (se construiește lista care reține deîmpărțitul). Apoi, cât timp gradul restului este mai mare sau egal cu gradul împărțitorului, se află monomul cu care se înmulțește câtul pentru a-l scădea din rest, se calculează produsul dintre monomul astfel determinat și cât și se scade polinomul astfel rezultat din rest, obținându-se un nou rest. Și aici, listele conținând polinoame care nu mai sunt necesare, se sterg.

Varianta Pascal	Varianta C++
<pre>procedure divp(deim,imp:adrNod; var cat,rest:adrNod); var coef:real; gradul:integer; sfcat,sfreast,pol,neg, factor,sfactor:adrNod; begin cat:=nil; rest:=nil; while deim<>nil do begin adaug(rest,sfreast, deim^.grad,deim^.coef); deim:=deim^.adr_urm; end; while rest^.grad>=imp^.grad do begin coef:=rest^.coef/imp^.coef; gradul:=rest^.grad-imp^.grad; adaug(cat,sfcat,gradul,coef); factor:=nil; end; end;</pre>	<pre>void divp(Nod *deim, Nod *imp, Nod*& cat, Nod*& rest) { float coef; int gradul; Nod* sfcat=0,*sfrest=0; cat=rest=0; while (deim) { adaug(rest,sfreast, deim->grad,deim->coef); deim=deim->adr_urm; } while (rest->grad>=imp->grad) { coef= rest->coef/ imp->coef; gradul=rest->grad- imp->grad; adaug(cat,sfcat, gradul,coef); Nod *factor=0, *sfactor=0; adaug(factor,sfactor, gradul,coef); } }</pre>

<pre>adaug(factor,sfactor, gradul,coef); pol:=mul(imp,factor); sterg(factor); neg:=negativ(pol); sterg(pol); rest:=adun(rest,neg); sterg(neg); end; end;</pre>	<pre>Nod* pol=mul(imp,factor); sterg(factor); Nod* neg=negativ(pol); sterg(pol); rest=adun(rest,neg); sterg(neg); } }</pre>
---	---

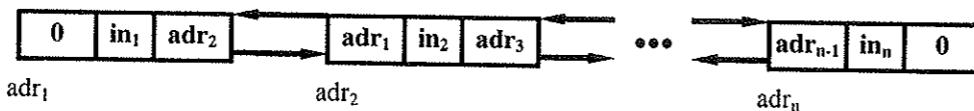
Pentru a testa subprogramele prezentate, puteți folosi programul următor, care citește două polinoame și calculează suma, diferența și produsul lor. De asemenea, se calculează și câtul și restul împărțirii celor două polinoame.

Varianta Pascal	Varianta C++
<pre>type adrNod=^Nod; Nod=record coef:real; grad:integer; adr_urm:adrNod; end; var m,n:integer; p1,p2,s,d,p,cat,rest:adrNod; { subprogramele prezentate } begin write('Nr termeni primul polinom='); readln(m); p1:=crePolinom(m); write('Nr termeni al doilea polinom='); readln(n); p2:=crePolinom(n); { suma } s:=adun(p1,p2); writeln('Suma este'); afis(s); { diferența } d:=adun(p1,negativ(p2)); writeln('diferența este'); afis(d); { produs } p:=mul(p1,p2); writeln('produsul este'); afis(p); { cat } divp(p1,p2,cat,rest); writeln('câtul este'); afis(cat); writeln('rest este'); if rest=nil then writeln(0) else afis(rest); end.</pre>	<pre>struct Nod { float coef; int grad; Nod* adr_urm; }; // subprogramele prezentate main() { int m,n; cout<<"Nr termeni primul polinom="; cin>>m; Nod* p1=crePolinom(m); cout<<"Nr termeni al doilea polinom="; cin>>n; Nod* p2=crePolinom(n); // suma Nod* s=adun(p1,p2); cout<<"Suma este"<<endl; afis(s); // diferența Nod* d=adun(p1,negativ(p2)); cout<<"diferența este"<<endl; afis(d); // produs Nod* p=mul(p1,p2); cout<<"produsul este"<<endl; afis(p); // cât Nod* cat,*rest; divp(p1,p2,cat,rest); cout<<"câtul este"<<endl; afis(cat); cout<<"rest este"<<endl; if (rest==0) cout<<0; else afis(rest); }</pre>

2.3. Liste liniare alocate dublu înlățuit



Definiția 2.3. O listă alocată dublu înlățuit este o structură de date de formă:



Avantajul utilizării listei alocate dublu înlățuit este dat de faptul că o astfel de listă poate fi parcursă în ambele sensuri.

Operațiile pe care le facem cu o listă dublu înlățuită sunt următoarele:

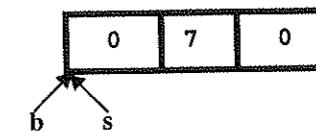
- 1) creare;
- 2) adăugare la dreapta;
- 3) adăugare la stânga;
- 4) adăugare în interiorul listei;
- 5) ștergere din interiorul listei;
- 6) ștergere la stânga listei;
- 7) ștergere la dreapta listei;
- 8) listare de la stânga la dreapta;
- 9) listare de la dreapta la stânga.

2.3.1. Crearea unei liste liniare alocate dublu înlățuit

O listă dublu înlățuită se creează cu o singură înregistrare. Pentru a ajunge la numărul de înregistrări dorit, utilizăm funcții de adăugare la stânga sau la dreapta. În programul de față acest lucru este realizat de funcția **creare**. Această funcție realizează operațiile următoare:

- citirea informației numerice;
- alocarea de spațiu pentru înregistrare;
- completarea înregistrării cu informația numerică;
- completarea adreselor de legătură la stânga și la dreapta cu 0;
- variabilele tip referință **b** și **s** vor căpăta valoarea adresei acestei prime înregistrări (**b** semnifică adresa înregistrării cea mai din stânga, **s** adresa ultimei înregistrări din dreapta).

Ex: Creăm lista cu un singur element:

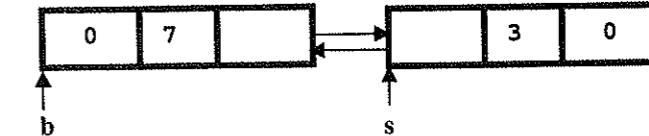


2.3.2. Adăugarea unei înregistrări la dreapta

Această operație este realizată de funcția **Addr**. Pentru adăugarea unei înregistrări se realizează următorii pași:

- citirea informației numerice;
- alocarea spațiului pentru înregistrare;
- completarea adresei la dreapta cu 0;
- completarea adresei din stânga cu adresa celei mai din dreapta înregistrări (reținute în variabila **s**);
- modificarea câmpului de adresă la dreapta a înregistrării din **s** cu adresa noii înregistrări;
- **s** va lua valoarea noii înregistrări, deoarece aceasta va fi cea mai din dreapta.

Ex: Adăugăm la dreapta înregistrarea 3.



2.3.3. Adăugarea unei înregistrări la stânga

Ex: Această operație vă este propusă ca exercițiu!

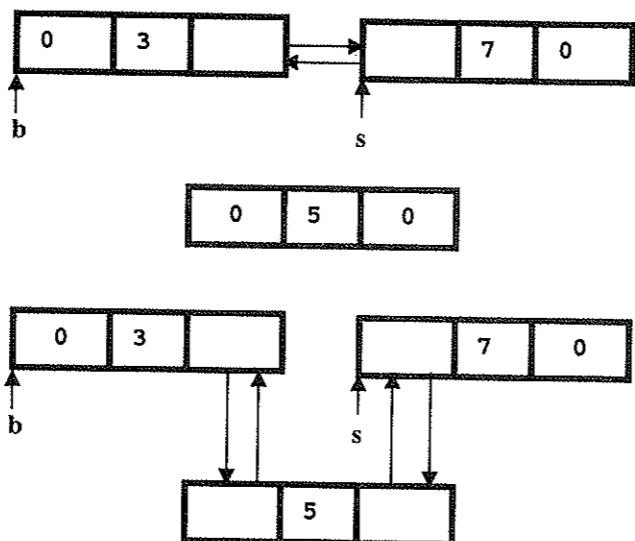
2.3.4. Adăugarea unei înregistrări în interiorul listei

Această operație este realizată de funcția **Includ**:

- parcurge lista de la stânga la dreapta căutând înregistrarea cu informația numerică **m**, în dreapta căreia urmează să introducем noua înregistrare;
- citește informația numerică;
- alocă spațiu pentru noua înregistrare;

- completează informația utilă;
- adresa stângă a noii înregistrării ia valoarea adresei înregistrării de informație utilă m ;
- adresa stângă a înregistrării care urma la acest moment înregistrării cu informația numerică m capătă valoarea adresei noii înregistrării;
- adresa dreaptă a noii înregistrării ia valoarea dreptă a înregistrării cu informația utilă m ;
- adresa dreaptă a înregistrării cu informația numerică m ia valoarea noii înregistrării.

Ex: În lista următoare adăugăm, după înregistrarea 3, înregistrarea 5:



Propunem ca exercițiu realizarea unei funcții de adăugare în interiorul listei a unei înregistrări la stânga înregistrării cu informația numerică m .

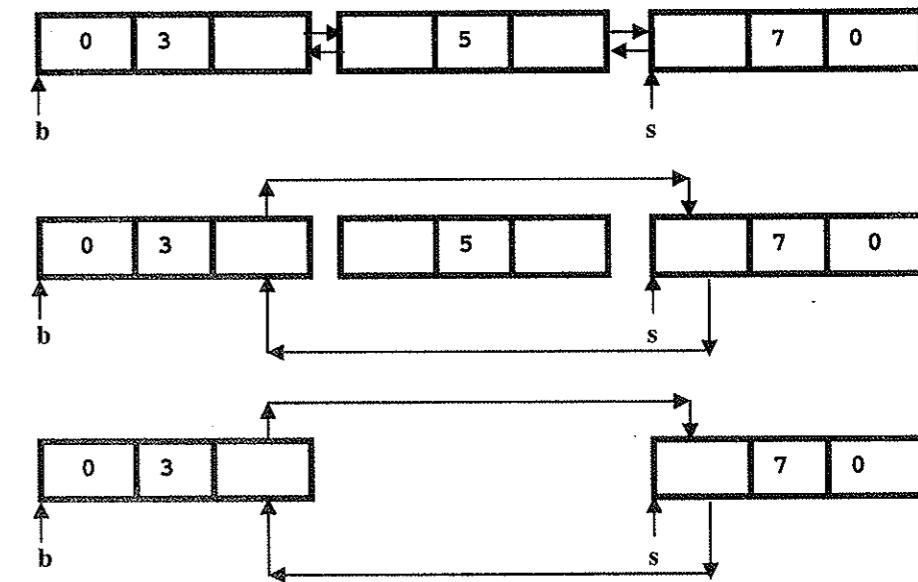
2.3.5. Ștergerea unei înregistrări din interiorul listei

Această operație este realizată de funcția **sterg**. Operațiile efectuate de această funcție sunt următoarele:

- se parurge lista de la stânga la dreapta pentru a ne poziționa pe înregistrarea care urmează a fi ștersă;
- câmpul de adresă dreaptă al înregistrării care o precede pe aceasta va lua valoarea câmpului de adresă dreaptă al înregistrării care va fi ștersă;

- câmpul de adresă stânga al înregistrării care urmează înregistrării care va fi ștersă va lua valoarea câmpului de adresă stânga al înregistrării pe care o ștergem;
- se eliberează spațiul de memorie rezervat înregistrării care se șterge.

Ex: În lista de mai jos se șterge elementul cu informația numerică 5:



2.3.6. Ștergerea unei înregistrări la stânga/dreapta listei

Acste două operații sunt propuse ca exerciții!

2.3.7. Listarea de la stânga la dreapta listei

Această operație este realizată de funcția **Listare** care realizează următoarele operații:

- pornește din stânga listei;
- atât timp cât nu s-a ajuns la capătul din dreapta al listei, se tipărește informația numerică și se trece la înregistrarea următoare.

2.3.8. Listarea de la dreapta la stânga listei

Operația se propune ca exercițiu!

În continuare sunt prezentate subprogramele și un exemplu de utilizare a lor:

Varianta Pascal	Varianta C++
<pre> type ref=^inr; inr=record as:ref; nr:integer; ad:ref end; var b,s,c:ref; n,m,i:integer; procedure creare(var b,s:ref); begin write('n='); readln(n); new(b); b^.nr:=n; b^.as:=nil; b^.ad:=nil; s:=b end; procedure addr(var s:ref); var d:ref; begin write('n='); readln(n); new(d); d^.nr:=n; d^.as:=s; d^.ad:=nil; s^.ad:=d; s:=d end; procedure listare(b:ref); var d:ref; begin d:=b; while d<>nil do begin writeln(d^.nr); d:=d^.ad end end; procedure includ (m:integer;b:ref); var d,e:ref; begin d:=b; while d^.nr<>m do d:=d^.ad; write('n='); readln(n); new(e); e^.nr:=n; e^.as:=d; d^.ad^.as:=e; e^.ad:=d^.ad; d^.ad:=e end; </pre>	<pre> #include <iostream.h> struct Nod { Nod *as, *ad; int nr; }; Nod *b,*s,*c; int n,m,i; void Creare (Nod*& b, Nod*& s) { cout<<"n="; cin>>n; b=new Nod; b->nr=n; b->as=b->ad=0; s=b; } void Addr(Nod*& s) { cout<<"n="; cin>>n; Nod* d=new Nod; d->nr=n; d->as=s; d->ad=0; s->ad=d; s=d; } void Listare(Nod*& b) { Nod* d=b; while (d) { cout<<d->nr<<endl; d=d->ad; } } void Includ(int m, Nod* b) { Nod *d=b, *e; while (d->nr!=m) d=d->ad; cout<<"n="; cin>>n; e=new Nod; e->nr=n; e->as=d; d->ad->as=e; e->ad=d->ad; d->ad=e; } void Sterg(int m, Nod* b) { Nod* d=b; while (d->nr!=m) d=d->ad; d->as->ad=d->ad; d->ad->as=d->as; delete d; } </pre>

```

procedure sterg (m:integer;
  b:ref);
var d:ref;
begin
  d:=b;
  while d^.nr>m do d:=d^.ad;
  d^.as^.ad:=d^.ad;
  d^.ad^.as:=d^.as;
  dispose(d)
end;

begin
  writeln('Creare lista cu o singura inregistr. ');
  Creare (b,s);
  cout<<"Cate inregistrari se adauga ?"; cin>>m;
  for (i=1;i<=m;i++) Addr(s);
  cout<<"Acum listez de la stanga la dreapta"<<endl;
  Listare(b);
  cout<<"Includem la dreapta o inregistrare "<<endl;
  cout<<"dupa care inregistrare se face includerea? "; cin>>m;
  Includ (m,b);
  cout<<"Acum listez de la stanga la dreapta"<<endl;
  Listare(b);
  cout<<"Acum stergem o inregistrare din interior"<<endl;
  cout<<"Ce inregistrare se sterge? ";
  cin>>m;
  Sterg(m,b);
  cout<<"Acum listez de la stanga la dreapta"<<endl;
  Listare(b);
}

```

2.4. Stiva implementată ca listă liniară simplu înlățuită

 **Definiția 2.4.** Stiva este o listă pentru care singurele operații permise sunt:

- adăugarea unui element în stivă;
- eliminarea, consultarea sau modificarea ultimului element introdus în stivă.

Stiva funcționează pe principiul **LIFO** (Last In First Out) - "ultimul intrat primul ieșit".

Analizați programul următor, care creează o stivă prin utilizarea unei liste liniare simplu înlățuite. Adăugarea unui element în stivă se face cu subprogramul **PUSH**, iar eliminarea, cu subprogramul **POP**. Vârful stivei este reținut de variabila **v**.

Varianta Pascal	Varianta C++
<pre> type Adresa=^Nod; Nod=record info:integer; adr_inap:Adresa; end; var v:Adresa; n:integer; procedure Push(var v:Adresa; n:integer); var c:Adresa; begin if v=nil then begin new(v); v^.info:=n; v^.adr_inap:=nil; end else begin new(c);c^.info:=n; c^.adr_inap:=v; v:=c; end end; procedure Pop (var v:Adresa); var c:Adresa; begin if v=nil then writeln('stiva este vida') else begin c:=v; writeln('am scos', c^.info); v:=v^.adr_inap; dispose(c) end; end; begin Push(v,1); Push(v,2); Push(v,3); Pop(v); Pop(v); Pop(v); Pop(v); end. </pre>	<pre> #include <iostream.h> struct Nod { int info; Nod* adr_inap; }; Nod* v; int n; void Push (Nod*& v,int n) { Nod* c; if (!v) { v= new Nod; v->info=n; v->adr_inap=0; } else { c= new Nod; c->info=n; c->adr_inap=v; v=c; } void Pop (Nod*& v) { Nod* c; if (!v) cout<<"stiva este vida"; else { c=v; cout<<"am scos" << c->info<<endl; v=v->adr_inap; delete c; } main() { Push(v,1); Push(v,2); Push(v,3); Pop(v); Pop(v); Pop(v); Pop(v); } </pre>

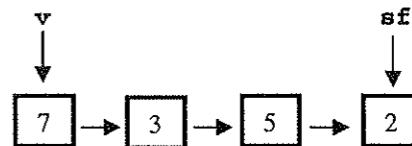
2.5. Coada implementată ca listă liniară simplu înlățuită



Definiția 2.5. O coadă este o listă pentru care toate inserările sunt făcute la unul din capete, iar toate ștergerile (consultările, modificările) la celălalt capăt.

Coada funcționează pe principiul **FIFO** (First In First Out) - "primul intrat primul ieșit".

Alocarea dinamică înlățuită a cozii. O variabilă **v** va reține adresa elementului care urmează a fi scos (servit). O altă, numită **sf**, va reține adresa ultimului element introdus în coadă. Figura următoare prezintă o coadă în care primul element care urmează a fi scos are adresa în **v**, iar ultimul introdus are adresa în **sf**.



Varianta Pascal	Varianta C++
<pre> type Adresa=^Nod; Nod=record info:integer; adr_urm:Adresa; end; var v,sf:Adresa; n:integer; procedure Pune(var v, sf:Adresa;n:integer); var c:Adresa; begin if v=nil then begin new(v); v^.info:=n; v^.adr_urm:=nil; sf:=v; end else begin new(c); sf^.adr_urm:=c; c^.info:=n; c^.adr_urm:=nil; sf:=c end end; procedure Scoate(var v, sf:Adresa); var c:Adresa; begin if v=nil then writeln('coada este vida') else begin c:=v; v:=v^.adr_urm; dispose(c) end; end; </pre>	<pre> #include<iostream.h> struct Nod { int info; Nod* adr_urm; }; Nod* v,*sf; int n; void Pune(Nod*& v,Nod*& sf,int n) { Nod* c; if (!v) { v=new Nod; v->info=n; v->adr_urm=0; sf=v; } else { c=new Nod; sf->adr_urm=c; c->info=n; c->adr_urm=0; sf=c; } } void Scoate(Nod*& v) { Nod* c; if (!v) cout<<"coada este vida"; else { cout<<"Am scos " <<v->info<<endl; c=v; v=v->adr_urm; delete c; } } </pre>

```

{ subprogram de Listare a
elementelor aflate in coada }
begin
Pune(v,sf,1); Pune(v,sf,2);
Pune(v,sf,3); Listare(v);
Scoate(v,sf); Listare(v);
Scoate(v,sf); Listare(v);
Scoate(v,sf); Listare(v);
end.

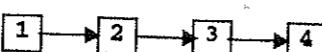
// subprogram de Listare a
//elementelor aflate in coada
main()
{ Pune(v,sf,1); Pune(v,sf,2);
Pune(v,sf,3); Listare(v);
Scoate(v); Listare(v);
Scoate(v); Listare(v);
Scoate(v); Listare(v);
Scoate(v); Listare(v);
}
  
```

Probleme propuse

1. Asociați fiecărui tip de listă denumit în coloana din stânga desenul corespunzător din coloana aflată în partea dreaptă a tabelului următor, scriind cifra asociată fiecărei litere:

A. listă liniară simplu înlățuită	1.	
B. listă neliniară simplu înlățuită	2.	
C. listă liniară dublu înlățuită	3.	
D. listă neliniară dublu înlățuită	4.	

2. Care dintre nodurile listei următoare (identificate prin numerele între 1 și 4) este primul element al listei?



- a) 1; b) 4; c) 2; d) nu există un prim element.

3. Dacă o listă formată din două noduri (identificate prin numerele 1 și 2) are proprietatea că elementul următor nodului 2 este nodul 1 și nu există un element următor nodului 1, atunci spunem că lista:

- a) este circulară; b) este liniară simplu înlățuită;
c) nu este liniară; d) este liniară dublu înlățuită.

4. Știind că adresa de început a listei reprezentate în desenul următor este memorată în variabila **p** și că fiecare nod al listei reține în câmpul **inf** numărul scris în desen și în câmpul **adr_urm** adresa elementului următor, stabiliți ce reprezintă expresia de mai jos.

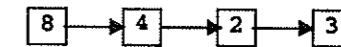
Varianță Pascal	Varianță C++
p^.adr_urm^.adr_urm^.nr	p->adr_urm->adr_urm->nr



- a) este o expresie incorectă;
b) valoarea memorată în nodul al treilea (valoarea 2);
c) adresa elementului al treilea (elementului ce memorează valoarea 2);
d) valoarea memorată în nodul al doilea (valoarea 4);
e) adresa elementului al doilea (elementului ce memorează valoarea 4).

5. Știind că adresa de început a listei reprezentate în desenul următor este memorată în variabila **p** și că fiecare nod al listei reține în câmpul **inf** numărul scris în desen și în câmpul **adr_urm** adresa elementului următor, stabiliți ce reprezintă expresia:

Varianță Pascal	Varianță C++
p^.adr_urm^.nr^.adr_urm	p->adr_urm->nr->adr_urm



- a) este o expresie incorectă;
b) valoarea memorată în nodul al doilea (valoarea 4);
c) adresa elementului al treilea (elementului ce memorează valoarea 2);
d) valoarea memorată în nodul al treilea (valoarea 2).

6. Știind că există o listă liniară simplu înlățuită nevidă, fiecare nod reținând în câmpul **ref** adresa elementului următor al listei, și știind că variabilele **v** și **s** rețin adresa primului și respectiv adresa ultimului element al listei, explicați care este efectul instrucțiunii:

Varianță Pascal	Varianță C++
s^.ref=v	s->ref=v

7. Știind că există o listă liniară simplu înlățuită cu cel puțin două noduri, fiecare nod reținând în câmpul **urm** adresa elementului următor al listei, și știind că variabilele **ini** și **fin** rețin adresa primului și respectiv adresa ultimului element al listei, explicați care este efectul instrucțiunii:

Varianța Pascal	Varianța C++
ini^.urm=fin	ini->urm=fin

8. Scrieți un subprogram care creează o listă liniară simplu înlățuită în care, fiecare nod, pe lângă informația de adresă, va conține o variabilă de tip **Struct** care reține date referitoare la un elev. Funcția va returna adresa primului nod al listei. Datele se citesc de la tastatură.

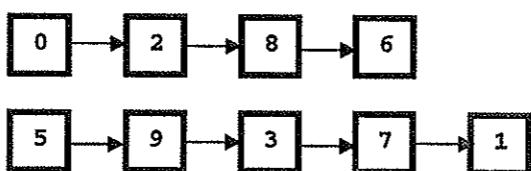
- numele: 20 de caractere;
- prenumele: 20 de caractere;
- un vector de numere reale cu 3 componente care rețin notele elevului.

9. Scrieți un subprogram care afișează pe monitor numele, prenumele și media generală a fiecărui elev din lista de mai sus. Funcția va primi ca parametru de intrare vârful listei.

10. Scrieți o funcție care returnează media generală a elevilor care se găsesc în listă.

11. Scrieți un program care creează și afișează o listă liniară simplu înlățuită. Fiecare nod al listei conține, pe lângă informația de adresă, un număr natural mai mic sau egal cu 100000. Numerele se găsesc, toate pe o linie, în ordine, separate prin un spațiu (blank) în fișierul text "lista.in".

12. Scrieți un program care creează și afișează două liste liniare simplu înlățuite. Prima listă va conține, în ordinea citirii, numere pare, iar a doua va conține, în aceeași ordine, numere impare. Numerele se citesc din fișierul text "numere.in". Ele se găsesc toate pe o linie și sunt separate prin cel puțin un spațiu. De exemplu, dacă fișierul text conține numerele: 0 2 5 9 8 3 6 7 1, atunci listele vor fi:



13. Scrieți un subprogram care creează fișierul text "liste.out" cu informațiile afilate în cele două liste de la problema anterioară. Prima linie va conține numerele din prima listă, a doua numerele din a doua listă. **Exemplu:** pentru listele din figura de mai sus, fișierul va fi:

Linia 1 0 2 8 6
Linia 2 5 9 3 7 1

14. Scrieți un subprogram care adaugă un nod la sfârșitul unei liste liniare simplu înlățuite. Fiecare nod al listei conține, pe lângă informația de adresă, un număr real. Subprogramul are ca parametri formali adresa primului element al listei și valoarea reală care se adaugă.

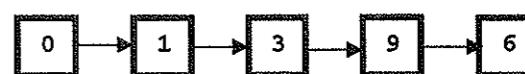
15. Scrieți o funcție care adaugă un nod la începutul unei liste liniare simplu înlățuite. Fiecare nod al listei conține, pe lângă informația de adresă, un număr real. Funcția are ca parametri formali adresa primului element al listei și valoarea reală care se adaugă. Ea returnează noua adresă de început a listei.

16. Se dă o listă liniară simplu înlățuită ale cărei noduri rețin, pe lângă informațiile de adresă, numere naturale cu o singură cifră. Lista are cel puțin un nod și cel mult 6 noduri. Se cere să se scrie o funcție care calculează și afișează valoarea întreagă obținută prin lipirea cifrelor memorate în listă în ordinea citirii. Funcția va primi ca parametru de intrare vârful listei. **Exemplu:** pentru lista de mai jos se afișează valoarea 956:



17. Prin operația de concatenare a două liste liniare simplu înlățuite se obține o a treia listă liniară simplu înlățuită care conține, în ordine, nodurile primei liste, urmate de nodurile celei de-a doua liste. Să se scrie o funcție care concatenează două liste date prin adresele nodurilor de început. Funcția va returna adresa primului nod al noii liste.

18. Se citește un fișier text **cifre.in** care conține, pe o unică linie, numai numere naturale între 0 și 9. Numerele nu sunt separate prin spații. Se cere să se formeze o listă liniară simplu înlățuită în care fiecare nod reține o cifră. **Exemplu:** Pentru 01396 se obține lista:



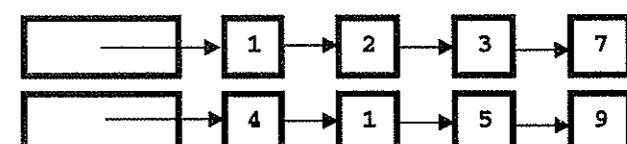
19. Să se scrie o funcție care primește ca parametru de intrare adresa unei liste liniare simplu înlățuite și are rolul de a inversa nodurile aflate pe prima și ultima poziție. Funcția va returna adresa primului nod al listei.

20. Scrieți un subprogram care eliberează spațiul ocupat de o listă liniară simplu înlățuită.

21. Scrieți un subprogram care memorează un tablou bidimensional cu **m** linii și **n** coloane ca **m** liste liniare simplu înlățuite, unde fiecare listă memorează, în ordine, elementele unei linii.

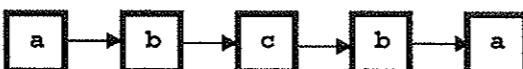
Exemplu: pentru tabloul următor se obțin listele:

[1 2 3 7]
[4 1 5 9]

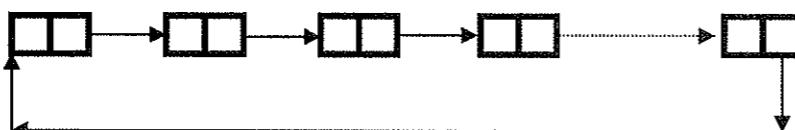


22. Se dă o listă liniară simplu înlățuită în care fiecare nod reține un caracter. Să se scrie o funcție care depistează dacă lista conține caractere distincte sau nu. Ce valoare trebuie să întoarcă o astfel de funcție?

23. Se dă o listă liniară simplu înlățuită în care fiecare nod reține o literă. Se cere să se scrie o funcție care depistează dacă cuvântul format prin alăturarea literelor citite este sau nu palindrom (se obține același rezultat dacă cuvântul se citește direct sau invers). De exemplu, lista următoare conține un cuvânt palindrom.



24. În cazul în care pentru o listă liniară simplu înlățuită câmpul de adresă al ultimului nod reține adresa primului nod, se obține o **listă circulară**:



Creați o listă circulară în care fiecare nod reține un număr natural. De asemenea, scrieți subprograme de inserare și ștergere a unui nod al listei create.

25. Se citește o permutare a numerelor $1, 2, \dots, n$. Se cere ca, prin utilizarea unei liste circulare, să se afișeze toate permutările circulare ale acesteia.

Exemplu: Se citește 1 2 3. Se va afișa: 1 2 3, 2 3 1, 3 1 2.

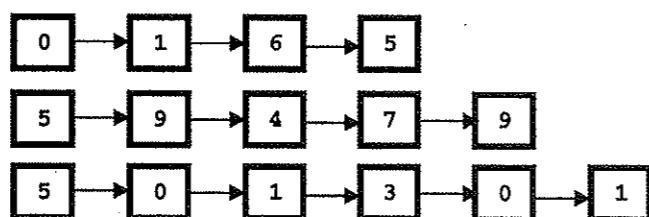
26. Scrieți un subprogram care transformă o listă liniară simplu înlățuită în una dublu înlățuită.

27. După cum știți, nu se poate lucra **în mod direct** cu numere naturale oricât de mari. Din acest motiv, vom memora un număr natural ca o listă liniară simplu înlățuită. De exemplu, numărul 5610 se va memora sub forma de mai jos. Scrieți un subprogram care citește de la tastatură cifrele unui număr natural, începând cu cifra cea mai semnificativă, și-l memorează ca listă liniară simplu înlățuită.

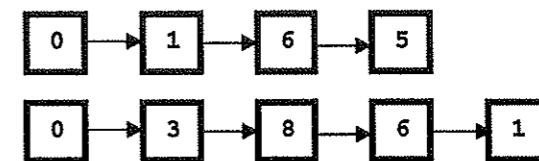


28. Scrieți un subprogram recursiv care primește ca parametru de intrare adresa unei liste liniare simplu înlățuite care reține un număr ca mai sus și afișează numărul. De exemplu, pentru lista de mai sus se afișează 5610.

29. Scrieți o funcție care adună două numere naturale memorate ca mai sus și returnează adresa de început a numărului sumă, memorat ca listă. **Exemplu:** din primele liste rezultă a treia listă:



30. Scrieți o funcție care calculează produsul dintre un număr natural memorat sub formă de listă și un altul, cu o singură cifră, transmis ca parametru. Funcția returnează adresa primului nod al listei care conține rezultatul. **Exemplu:** numărul de mai jos se înmulțește cu 3 și rezultă:



31. Scrieți o funcție care înmulțește două numere naturale memorate sub formă de liste și returnează adresa de început a listei rezultat.

32. **Lucrare în echipă.** Scrieți un ansamblu de subprograme (numim ansamblul **NUMERE_MARI**) care să ne ajute să lucrăm cu numere **întregi**, oricât de mari, memorate sub formă de liste. Utilizatorul poate efectua adunarea, scăderea, înmulțirea și împărțirea a două astfel de numere. De asemenea, vor exista funcții care să permită efectuarea comparațiilor între două numere: mai mare, mai mic, egal, mai mare sau egal, mai mic sau egal.

33. Prin utilizarea asamblului numit **NUMERE_MARI**, calculați maximul a n numere întregi, citite de la tastatură.

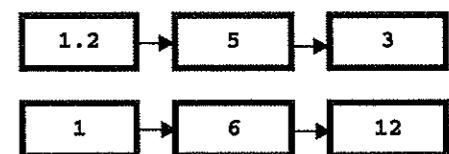
34. Prin utilizarea asamblului numit **NUMERE_MARI**, sortați crescător n numere întregi, citite de la tastatură.

35. Prin utilizarea asamblului numit **NUMERE_MARI**, calculați $n!$, unde n este citit de la tastatură.

36. Prin utilizarea asamblului numit **NUMERE_MARI**, calculați: $\sum_{k=1}^n k!$

37. **Lucrare în colectiv.** Se numește **matrice rară** o matrice în care majoritatea elementelor sunt nule. O **matrice rară** va fi memorată cu ajutorul a două liste liniare simplu înlățuite, una conținând valorile nenule, alta numărul de ordine al lor, număr rezultat prin parcurgerea pe linii a matricei. Scrieți un set de subprograme cu ajutorul căror să se poată efectua cu ușurință operații cu matrice rare: adunare, scădere, înmulțire. **Exemplu:** matricea următoare se memorează așa cum se vede:

$$A = \begin{pmatrix} 1.2 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$



Răspunsuri la testele grilă

1. Se realizează asocierile: A-3, B-1, C-2, D-4; 2. a); 3. b); 4. b); 5. a).

Capitolul 3

Metoda DIVIDE ET IMPERA

3.1. Prezentare generală

DIVIDE ET IMPERA este o tehnică specială și se bazează pe un principiu extrem de simplu: descompunem problema în două sau mai multe subprobleme (mai ușoare), care se rezolvă, iar soluția pentru problema inițială se obține combinând soluțiile problemelor în care a fost descompusă. Se presupune că fiecare dintre problemele în care a fost descompusă problema inițială, se poate descompune în alte subprobleme, la fel cum a fost descompusă problema inițială. Procedeul se reia până când (în urma descompunerilor repetate) se ajunge la probleme care admit rezolvare imediată.

Evident, nu toate problemele pot fi rezolvate prin utilizarea acestei tehnici. Fără teamă de a greși, putem afirma că numărul lor este relativ mic, tocmai datorită cerinței ca problema să admită o descompunere repetată.

DIVIDE ET IMPERA este o tehnică ce admite o implementare recursivă. Am învățat principiul general prin care se elaborează algoritmii recursivi: ce se întâmplă la un nivel, se întâmplă la orice nivel (având grija să asigurăm condițiile de terminare). Tot așa, se elaborează un algoritm prin **DIVIDE ET IMPERA**. La un anumit nivel, avem două posibilități:

- 1) am ajuns la o problemă care admite o rezolvare imediată, caz în care se rezolvă și se revine din apel (condiția de terminare);
- 2) nu am ajuns în situația de la punctul 1, caz în care descompunem problema în două sau mai multe subprobleme, pentru fiecare din ele reapelăm funcția, combinăm rezultatele și revenim din apel.

3.2. Aplicații

3.2.1. Valoarea maximă dintr-un vector

Problema 3.1. Se citește un vector cu **n** componente, numere naturale. Se cere să se tipărească valoarea maximă.

Problema de mai sus este binecunoscută. Cum o rezolvăm utilizând tehnica **DIVIDE ET IMPERA**?

Rezolvare. Trebuie tipărită valoarea maximă dintre numerele reținute în vector de la **i** la **j** (inițial, **i=1** și **j=n**).

Pentru aceasta, procedăm astfel:

- dacă **i=j**, valoarea maximă va fi **v[i]**;
- contrar, vom împărți vectorul în doi vectori (primul vector va conține componentele de la **i** la **(i+j) div 2**, al doilea va conține componentele de la **(i+j) div 2 + 1** la **j**, rezolvăm subproblemele (aflăm maximul pentru fiecare din ele) iar soluția problemei va fi dată de valoarea maximă dintre rezultatele celor două subprobleme.

Programul este următorul:

Varianta Pascal	Varianta C++
<pre>var v:array[1..10] of integer; n,i:integer; function max(i,j:integer) :integer; var a,b:integer; begin if i=j then max:=v[i] else begin a:=max(i,(i+j) div 2); b:=max((i+j) div 2+1,j); if a>b then max:=a else max:=b; end; end; begin write('n='); readln(n); for i:=1 to n do begin write('v[',i,']='); readln(v[i]); end; writeln('max=',max(1,n)); end.</pre>	<pre>#include <iostream.h> int v[10],n; int max(int i,int j) { int a,b; if (i==j) return v[i]; else { a=max(i,(i+j)/2); b=max((i+j)/2+1,j); if (a>b) return a; else return b; } } main() { cout<<"n="; cin>>n; for (int i=1;i<=n;i++) { cout<<"v["<<i<<"]="; cin>>v[i]; } cout<<"max="<<max(1,n); }</pre>

 Algoritmul prezentat este exclusiv didactic, în practică este preferat algoritmul clasic.

3.2.2. Sortarea prin interclasare

Problema 3.2. Se consideră vectorul a cu n componente numere întregi (sau reale). Să se sorteze crescător, utilizând sortarea prin interclasare.

Interclasarea a doi vectori a fost studiată. Dacă dispunem de două siruri de valori, primul cu m elemente, al doilea cu n elemente, ambele sortate, atunci se poate obține un vector care conține *toate* valorile sortate. Algoritmul de interclasare este performant, pentru că efectuează *cel mult* $m+n-1$ comparații.

În cele ce urmează, vom utiliza algoritmul de interclasare în vederea sortării unui vector prin interclasare.

Rezolvare. Algoritmul de sortare prin interclasare se bazează pe următoarea idee: pentru a sorta un vector cu n elemente îl împărțim în doi vectori care, odată sortați, se interclasează.

Conform strategiei generale **DIVIDE ET IMPERA**, problema este descompusă în alte două subprobleme de același tip și, după rezolvarea lor, rezultatele se combină (în particular se interclasează). Descompunerea unui vector în altii doi vectori care urmează a fi sortați are loc până când avem de sortat vectori de una sau două componente.

În aplicație, funcția `sort` sortează un vector cu maximum două elemente; `interc` interclasează rezultatele; `divimp` implementează strategia generală a metodei studiate.

Varianța Pascal	Varianța C++
<pre>type vector=array [1..10] of integer; var a:vector; n,i:integer; procedure sort(p,q:integer; var a:vector); var m:integer; begin if a[p]>a[q] then begin m:=a[p]; a[p]:=a[q]; a[q]:=m; end end;</pre>	<pre>#include <iostream.h> int a[10],n; void sort(int p,int q, int a[10]) { int m; if (a[p]>a[q]) { m=a[p]; a[p]:=a[q]; a[q]:=m; } } void interc(int p,int q, int m,int a[10]) { int b[10],i,j,k; i=p; j=m+1; k=1;</pre>

```

procedure interc
(p,q,m:integer; var a:vector);
var b:vector;
i,j,k:integer;
begin
i:=p; j:=m+1; k:=1;
while (i<=m) and (j<=q) do
  if a[i]<=a[j] then
    begin
      b[k]:=a[i];
      i:=i+1; k:=k+1
    end
  else
    begin
      b[k]:=a[j];
      j:=j+1; k:=k+1
    end;
  if i<=m then
    for j:=i to m do
      begin
        b[k]:=a[j]; k:=k+1
      end
  else
    for i:=j to q do
      begin
        b[k]:=a[i]; k:=k+1
      end;
k:=1;
for i:=p to q do
begin
  a[i]:=b[k]; k:=k+1
end;
end;

procedure divimp
(p,q:integer;var a:vector);
var m:integer;
begin
if (q-p)<=1 then sort(p,q,a)
else
begin
  m:=(p+q) div 2;
  divimp(p,m,a);
  divimp(m+1,q,a);
  interc(p,q,m,a)
end;
end;

main()
{ int i;
cout<<"n="; cin>>n;
for (i=1;i<=n;i++)
{ cout<<"a["<<i<<"]=";
  cin>>a[i];
}
divimp(1,n,a);
for (i=1;i<=n;i++)
  cout<<a[i]<<" ";
}
```

În continuare, calculăm numărul aproximativ de comparații efectuat de algoritm. Fie acesta $T(n)$. Mai simplu, presupunem $n=2^k$.

O problemă se descompune în alte două probleme, fiecare cu $n/2$ componente, după care urmează interclasarea lor, care necesită $n/2+n/2=n$ comparații:

$$T(n) = \begin{cases} 0, & n = 1; \\ 2T\left(\frac{n}{2}\right) + n, & \text{altfel.} \end{cases}$$

Avem:

$$\begin{aligned} T(n) &= T(2^k) = 2(T(2^{k-1}) + 2^{k-1}) = 2T(2^{k-1}) + 2^k = 2T(2^{k-2} + 2^{k-1}) + 2^k = \\ &= 2T(2^{k-2}) + 2^k + 2^k = \dots \underbrace{2^k + 2^k + \dots + 2^k}_{\text{de } k \text{ ori}} = \underbrace{n + n + \dots + n}_{\text{de } k \text{ ori}} = n \cdot k = n \cdot \log_2 n \end{aligned}$$

3.2.3. Sortarea rapidă

Problema 3.3. Fie vectorul a cu n componente numere întregi (sau reale). Se cere ca vectorul să fie sortat crescător.

Rezolvare. Este necesară o funcție **POZ** care tratează o porțiune din vector, cuprinsă între indicii dați de li (limita inferioară) și ls (limita superioară). Rolul acestei funcții este de a poziționa prima componentă $a[li]$ pe o poziție k cuprinsă între li și ls , astfel încât toate componentele vectorului cuprinse între li și $k-1$ să fie mai mici sau egale decât $a[k]$ și toate componentele vectorului cuprinse între $k+1$ și ls să fie mai mari sau egale decât $a[k]$.

În această funcție există două moduri de lucru:

- a) i rămâne constant, j scade cu 1;
- b) i crește cu 1, j rămâne constant.

Funcția este concepută astfel:

- inițial, i va lua valoarea li , iar j va lua valoarea ls (elementul care inițial se află pe poziția li se va găsi mereu pe o poziție dată de i sau de j);
- se trece în modul de lucru a);
- atât timp cât $i < j$, se execută:
 - dacă $a[i]$ este strict mai mare decât $a[j]$, atunci se inversează cele două numere și se schimbă modul de lucru;
 - i și j se modifică corespunzător modului de lucru în care se află programul;
 - k ia valoarea comună a lui i și j .

 **Ex:** Pentru $a=(6,9,3,1,2)$, $li=1$, $ls=5$; modul de lucru a):

- $i=1$, $j=5$;
- $a[1] > a[5]$, deci se inversează elementele aflate pe pozițiile 1 și 5, deci $a=(2,9,3,1,6)$ și programul trece la modul de lucru b);
- $i=2$, $j=5$;
- $a[2] > a[5]$, deci $a=(2,6,3,1,9)$ și se revine la modul de lucru a);
- $i=2$, $j=4$;
- $a[2] > a[4]$, deci $a=(2,1,3,6,9)$; se trece la modul de lucru b);
- $i=3$, $j=4$;
- funcția se încheie, elementul aflat inițial pe poziția 1 se găsește acum pe poziția 4, toate elementele din stânga lui fiind mai mici decât el, totodată toate elementele din dreapta lui fiind mai mari decât el ($k=4$).

Alternanța modurilor de lucru se explică prin faptul că elementul care trebuie poziționat se compară cu un element aflat în dreapta sau în stânga lui, ceea ce impune o modificare corespunzătoare a indicilor i și j .

 După aplicarea funcției **POZ**, este evident că elementul care se află inițial în poziția li va ajunge pe o poziție k și va rămâne pe acea poziție în cadrul vectorului deja sortat, fapt care reprezintă esența algoritmului.

Funcția **QUICK** are parametrii li și ls (limita inferioară și limita superioară). În cadrul ei se utilizează metoda **DIVIDE ET IMPERA**, după cum urmează:

- se apelează **POZ**;
- se apelează **QUICK** pentru li și $k-1$;
- se apelează **QUICK** pentru $k+1$ și ls .

Varianta Pascal	Varianta C++
<pre>type vector=array [1..100] of integer; var i,n,k:integer; a:vector; procedure poz (li,ls:integer; var k:integer; var a:vector); var i,j,c,i1,j1:integer; begin i1:=0; j1:=-1; i:=li; j:=ls;</pre>	<pre>#include <iostream.h> int a[100],n,k; void poz (int li,int ls,int& k,int a[100]) { int i=li,j=ls,c,i1=0,j1=-1; while (i<j) { if (a[i]>a[j]) { c=a[j]; a[j]=a[i]; a[i]=c; c=i1; i1=-j1; j1=-c; } i=i+i1; j=j+j1; } k=i;</pre>

```

while i<j do
begin
  if a[i]>a[j]
  then
    begin
      c:=a[j];
      a[j]:=a[i];
      a[i]:=c;
      c:=i;
      i:=~j;
      j:=~c
    end;
    i:=i+1;
    j:=j+1
  end;
k:=i
end;

procedure quick(li,ls:integer);
begin
  if li<ls
  then
    begin
      poz(li,ls,k,a);
      quick(li,k-1);
      quick(k+1,ls)
    end
  end;

begin
  write('n=');
  readln(n);
  for i:=1 to n do
  begin
    write('a[',i,']=');
    readln(a[i])
  end;
  quick(1,n);
  for i:=1 to n do
    writeln(a[i])
end.

```

Reținăti! Sortarea rapidă efectuează în medie $n \cdot \log_2 n$ operații.

Demonstrația necesită cunoștințe de matematică pe care nu le aveți la nivelul acestui an de studiu...

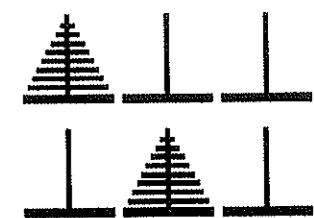
```

void quick (int li,int ls)
{ if (li<ls)
  { poz(li,ls,k,a);
    quick(li,k-1);
    quick(k+1,ls);
  }
}

main()
{ int i;
  cout<<"n="; cin>>n;
  for (i=1;i<=n;i++)
  { cout<<"a["<<i<<"]=";
    cin>>a[i];
  }
  quick(1,n);
  for (i=1;i<=n;i++)
    cout<<a[i]<<endl;
}

```

3.2.4. Turnurile din Hanoi



Problema 3.4. Se dau 3 tije simbolizate prin **a**, **b**, **c**.

Pe tija **a** se găsesc discuri de diametre diferite, așezate în ordine descrescătoare a diametrelor privite de jos în sus. Se cere să se mute discurile de pe tija **a** pe tija **b**, utilizând ca tijă intermedieră tija **c**, respectând următoarele reguli:

- la fiecare pas se mută un singur disc;
- nu este permis să se aşeze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

Rezolvare.

Dacă $n=1$, se face mutarea **ab**, adică se mută discul de pe tija **a** pe tija **b**.

Dacă $n=2$, se fac mutările **ac**, **ab**, **cb**.

În cazul în care $n > 2$, problema se complică. Notăm cu $H(n, a, b, c)$ sirul mutărilor celor n discuri de pe tija **a** pe tija **b**, utilizând ca tijă intermedieră, tija **c**.

Conform strategiei **DIVIDE ET IMPERA**, încercăm să descompunem problema în alte două subprobleme de același tip, urmând apoi combinarea soluțiilor. În acest sens, observăm că mutarea celor n discuri de pe tija **a** pe tija **b**, utilizând ca tijă intermedieră tija **c**, este echivalentă cu:

- mutarea a $n-1$ discuri de pe tija **a** pe tija **c**, utilizând ca tijă intermedieră tija **b**;
- mutarea discului rămas pe tija **b**;
- mutarea a $n-1$ discuri de pe tija **c** pe tija **b**, utilizând ca tijă intermedieră tija **a**.

Parcurserea celor trei etape permite definirea recursivă a sirului $H(n, a, b, c)$ astfel:

$$H(n, a, b, c) = \begin{cases} ab, & n = 1 \\ H(n-1, a, c, b), ab, H(n-1, c, b, a), & n > 1 \end{cases}$$

Ex: Prinții următoarele exemple:

1) pentru $n=2$, avem: $H(2, a, b, c) = H(1, a, c, b), ab, H(1, c, b, a) = ac, ab, cb$;

2) pentru $n=3$, avem:

$H(3, a, b, c) = H(2, a, c, b), ab, H(2, c, b, a) = H(1, a, b, c), ac, H(1, b, c, a), ab, H(1, c, a, b), cb, H(1, a, b, c) = ab, ac, bc, ab, ca, cb, ab$.

Varianta Pascal	Varianta C++
<pre> var a,b,c:char; n:integer; procedure han (n:integer; a,b,c:char); begin if n=1 then writeln(a,b) else begin han(n-1,a,c,b); writeln(a,b); han(n-1,c,b,a); end begin write('N='), readln(n); a:='a'; b:='b'; c:='c'; han(n,a,b,c) end. </pre>	<pre> #include <iostream.h> char a,b,c; int n; void han (int n,char a, char b,char c) { if (n==1) cout<<a<<b<<endl; else { han(n-1,a,c,b); cout<<a<<b<<endl; han(n-1,c,b,a); } main() { cout<<"N="; cin>>n; a='a'; b='b'; c='c'; han(n,a,b,c); } </pre>

3.2.5. Problema tăieturilor

Problema 3.5. Se dă o bucătă dreptunghiulară de tablă cu lungimea l și înălțimea h , având pe suprafața ei n găuri de coordonate numere întregi. Se cere să se decupeze din ea o bucătă de arie maximă care nu prezintă găuri. Sunt permise numai tăieturi verticale și orizontale.

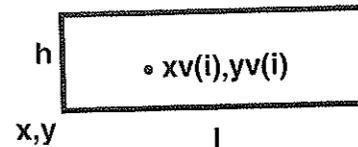
Rezolvare. Coordonatele găurilor sunt reținute în doi vectori xv și yv . Dreptunghiul inițial, precum și dreptunghiurile care apar în procesul tăierii sunt memorate în program prin coordonatele colțului din stânga-sus (x, y), prin lungime și înălțime (l, h).

Pentru un dreptunghi (initial pornim cu toată bucata de tablă), verificăm dacă avem sau nu o gaură în el (se caută practic prima din cele n găuri). În situația când acesta prezintă o gaură, problema se descompune în alte patru probleme de același tip. Dacă bucata nu prezintă găuri, se compară aria ei cu aria unei alte bucăți fără gaură, găsită în fazele precedente.

Mentionăm că dreptunghiul de arie maximă fără găuri este reținut prin aceeași parametri ca și dreptunghiul cu găuri, în zonele XF , YF , LF , HF .

În concluzie, problema inițială se descompune în alte patru probleme de același tip, mai ușoare, întrucât fiecare nou dreptunghi are cel mult $n-1$ găuri, dacă dreptunghiul inițial avea n găuri. La această problemă compararea soluțiilor constă în a reține dreptunghiul cu aria maximă dintre cele fără găuri.

Fie dreptunghiul cu o gaură:



Pentru a se afla în interiorul dreptunghiului, gaura trebuie să îndeplinească simultan condițiile:

- 1) $xv(i) > x$;
- 2) $xv(i) < x+l$;
- 3) $yv(i) > y$;
- 4) $yv(i) < y+h$.

Dacă facem o tăietură verticală prin această gaură, obținem două dreptunghiuri:

- 1) $x, y, xv(i)-x, h$;
- 2) $xv(i), y, l+x-xv(i), h$.

În urma unei tăieturi pe orizontală se obțin cele două dreptunghiuri:

- 1) $x, y, l, yv(i)-y$;
- 2) $x, yv(i), l, h+y-yv(i)$.

Programul este următorul:

Varianta Pascal	Varianta C++
<pre> type vect=array [1..9] of integer; var l,h,i,n,xf,yf, lf,hf:integer; xv,yv:vect; procedure dimp (x,y,l,h:integer; var xf,yf,lf,hf:integer; var xv,yv:vect); var gasit:boolean; i:integer; begin i:=1; gasit:=false; while (i<=n) and (not gasit) do if (xv[i]>x) and (xv[i]<l) and (yv[i]>y) and (yv[i]<y+h) then gasit:=true else i:=i+1; end; </pre>	<pre> #include <iostream.h> int l,h,i,n,xf,yf,lf, hf,xv[10],yv[10]; void dimp(int x,int y,int l, int h, int& xf, int& yf, int& lf,int& hf, int xv[10],int yv[10]) { int gasit=0,i=1; while (i<=n && !gasit) if (xv[i]>x && xv[i]<l && yv[i]>y && yv[i]<y+h) gasit=1; else i++; if (gasit) { dimp(x,y,xv[i]-x,xf, yf,lf,hf,xv,yv); dimp(xv[i],y,1+x-xv[i], h,xf,yf,lf,hf,xv,yv); dimp(x,y,1,yv[i]-y,xf, yf,lf,hf,xv,yv); dimp(x,yv[i],l,h+y-yv[i], xf,yf,lf,hf,xv,yv); } } </pre>

```

if gasit
then
begin
dimp(x,y,xv[i]-x,
      h,xf,yf,lf,hf,xv,yv);
dimp(xv[i],y,1+x-xv[i],
      h,xf,yf,lf,hf,xv,yv);
dimp(x,y,1,yv[i]-y,
      xf,yf,lf,hf,xv,yv);
dimp(x,yv[i],1,h+y-yv[i],
      xf,yf,lf,hf,xv,yv)
end
else
if (l*h)>(lf*hf)
then
begin
  xf:=x;
  yf:=y;
  lf:=l;
  hf:=h
end
end;
begin
write('n=');
readln(n);
for i:=1 to n do
begin
  write('x[',i,',']=');
  readln(xv[i]);
  write('y[',i,',']=');
  readln(yv[i])
end;
write('l=');
readln(l);
write('h=');
readln(h);
lf:=0;
hf:=0;
dimp(0,0,l,h,xf,yf,
      lf,hf,xv,yv);
writeln('x=',xf,' y=',yf,
      ' l=',lf,' h=',hf)
end.

```

3.3. Fractali

Fractalii au fost introdusi în anul 1975 prin lucrarea revoluționară a matematicianului francez Benoit Mandelbrot, "O teorie a serilor fractale", ce reunește totodată diversele teorii dinaintea sa. El este cel care a inventat cuvântul "fractal", de proveniență latină ("frângere" – a sparge în fragmente neregulate).

Noțiunea de **fractal** a apărut ca urmare a studiului vieții reale, în care informația genetică conținută în nucleul unei celule se repetă la diferite scări. Calculatorul permite ca o anumită figură (de exemplu, un segment) să se transforme într-o altă, formată din mai multe figuri inițiale (de exemplu, o linie frântă) și fiecare figură obținută să se transforme în mod asemănător (aceste transformări necesită foarte multe calcule).

Aceste forme geometrice au fost considerate în trecut haotice sau "aberații geometrice", iar multe dintre ele erau atât de complexe încât necesau calculatoare performante pentru a le vizualiza. Pe parcurs, domeniile științifice ca fizica, chimia, biologia sau meteorologia descopereau elemente asemănătoare cu fractalii în viața reală. Fractalii au proprietăți matematice extrem de interesante, care de multe ori contrazic aparența, dar acestea depășesc cu mult cunoștințele de matematică din liceu.

Înainte de a prezenta câteva exemple, trebuie cunoscute mai întâi noțiunile de bază pentru a lucra în mod grafic.

3.3.1. Elemente de grafică

3.3.1.1. Generalități (varianta Pascal)

Limbajul Pascal conține o serie de proceduri și funcții care permit realizarea unor aplicații grafice. Acestea sunt reunite în unitatea **GRAPH**, ce se găsește în subcatalogul **UNITS**.

Pentru ca o imagine să poată apărea pe ecran, calculatorul utilizează placa video, care diferă în funcție de memoria video și alți parametri. Pentru a accesa o placă video, trebuie să folosim anumite rutine speciale, specifice lor, numite **Driver-e**. Limbajul Pascal deține o colecție de astfel de componente software și în funcție de placă ce a fost detectată în sistem, se încarcă un driver sau altul. Aceste fișiere au extensia "**bgi**". Deoarece performanțele componentelor hardware au depășit cu mult capacitatele **CGA** sau **EGA**, ne vom referi în continuare doar la driver-ul **VGA** (**Video Graphics Array**), dezvoltat de firma IBM. Driver-ul **VGA** poate lucra în mai multe moduri, însă vom prefera modul standard de înaltă rezoluție "**VGAHI**" (constantă de tip întreg), ce poate afișa **640 x 480** puncte în 16 culori.

Selectarea driver-ului și a modului de lucru se face prin utilizarea procedurii **initgraph**. Aceasta are trei parametri: **gdriver** (de tip **integer**) care conține codul asociat driver-ului, **gmode** (de tip **integer**) care reține modul de lucru și o variabilă de tip **string**, care arată calea către unitatea **GRAPH**. Forma generală a acestei proceduri este

```
initgraph(gdriver,gmode,'cale');
```

Primii doi parametri sunt transmiși prin referință.

Inițializarea sistemului grafic se poate face în două feluri:

- 1) prin a solicita să se identifice automat placa grafică și corespunzător ei să se încarce un anumit driver și să se selecteze modul de lucru – cel mai bun din punct de vedere al performanțelor:

```
procedure initg;
begin
  gdriver := detect;
  initgraph(gdriver,gmode,'c:\tp\bgi');
  if graphresult<>0 then
    begin
      writeln("Tentativa esuata!");
      halt
    end
  end;
```

Constanta **detect** are valoarea 0 și se specifică procedurii identificarea automată a driver-ului și a modului de lucru.

Vom reține această procedură pentru că o vom utiliza în exemplele ulterioare.

- 2) prin indicarea cu ajutorul primilor doi parametri a unui driver și a unui mod de lucru solicitate de programator (în acest caz, nu se poate executa programul pe un calculator ce nu este dotat cu placa grafică specificată):

```
gdriver := VGA;
gmode := VGAHI;
initgraph(gdriver,gmode,'c:\tp\bgi');
if graphresult<>0 then
  begin
    writeln("Tentativa esuata!");
    halt
  end;
```

Tentativa de inițializare grafică poate eşua din diverse motive, cum ar fi: lipsa unității **GRAPH**, calea indicată greșit, etc. Testarea se realizează cu funcția întreagă **graphresult** care returnează 0 în caz afirmativ și o valoare diferită de 0, în caz contrar.

 Odată intrați în modul grafic, nu se mai poate scrie pe monitor, ca până acum (cu **write** sau **writeln**). Ieșirea din modul grafic se face prin utilizarea procedurii **closegraph**.

3.3.1.2. Generalități (varianta C++)

Limbajul C++ (în varianta Borland), conține o serie de funcții care permit realizarea unor aplicații grafice. Acestea sunt reunite în fișierul **GRAPHICS.H**, ce se găsește în folderul **INCLUDE**.

Pentru ca o imagine să poată apărea pe ecran, calculatorul utilizează placa video, care diferă în funcție de memoria video și alți parametri. Pentru a accesa o placă video, trebuie să folosim anumite rutine speciale, specifice lor, numite **Driver-e**. Limbajul C++ deține o colecție de astfel de componente software și în funcție de placa ce a fost detectată în sistem, se încarcă un driver sau altul. Aceste fișiere au extensia "bgi". Deoarece performanțele componentelor hardware au depășit cu mult capacitatele **CGA** sau **EGA**, ne vom referi în continuare doar la driver-ul **VGA** (**Video Graphics Array**), dezvoltat de firma IBM. Driver-ul VGA poate lucra în mai multe moduri, însă vom prefera modul standard de înaltă rezoluție "**VGAHI**" (constantă de tip întreg), ce poate afișa **640 x 480** puncte în 16 culori. Fișierul ce conține driver-ul utilizat este "**EGAVGA.CGI**".

Selectarea driver-ului și a modului de lucru se face prin utilizarea funcției **initgraph**. Aceasta are trei parametri: **gdriver** (de tip **integer**) care conține codul asociat driver-ului, **gmode** (de tip **integer**) care reține modul de lucru și o variabilă de tip **string**, care arată calea către unitatea **GRAPH**. Forma generală a acestei funcții este

```
initgraph(&gdriver,&gmode,"cale");
```

Primii doi parametri sunt transmiși prin referință.

Inițializarea sistemului grafic se poate face în două feluri:

- 1) prin a solicita să se identifice automat placa grafică și corespunzător ei să se încarce un anumit driver și să se selecteze modul de lucru – cel mai bun din punct de vedere al performanțelor:

```
void init()
{
  gdriver = DETECT;
  initgraph(&gdriver,&gmode,"E:\\BORLANDC\\BGI");
  if (graphresult())
  {
    cout<<"Tentativa nereusita.";
    cout<<"Apasa o tasta pentru a inchide...";
    getch();
    exit(1);
  }
}
```

Constanta **DETECT** are valoarea 0 și se specifică funcției identificarea automată a driver-ului și a modului de lucru.

Vom reține funcția **init()** pentru că o vom utiliza în exemplele ulterioare.

2) prin indicarea cu ajutorul primilor doi parametri a unui driver și a unui mod de lucru solicitate de programator (în acest caz, nu se poate executa programul pe un calculator ce nu este dotat cu placa grafică specificată):

```
gdriver := VGA; gmode := VGMHI;
initgraph(&gdriver,&gmode,"E:\\BORLANDC\\BGI");
if (graphresult())
{ cout<<"Tentativa nereusita.";
  cout<<"Apasa o tasta pentru a inchide..."; 
  getch();
  exit(1);
}
```

Tentativa de inițializare grafică poate eșua din diverse motive, cum ar fi: lipsa unității **GRAPHICS**, calea indicată greșit, etc. Testarea se realizează cu funcția întreagă **graphresult()** care returnează 0 în caz afirmativ și o valoare diferită de 0, în caz contrar.

! Odată intrați în modul grafic, nu se mai poate scrie pe monitor ca până acum (de exemplu, cu **cout**). Ieșirea din modul grafic se face prin utilizarea procedurii **closegraph()**.

Atenție! Pentru a putea scrie și rula programe C++ ce utilizează modul grafic al limbajului, trebuie bifată următoarea opțiune, din meniu:

Options / Linker / Libraries / Graphics library.

3.3.1.3. Setarea culorilor și procesul de desenare (Pascal și C++)

Cu siguranță, placa video utilizată de dvs. are performanțe superioare modului standard VGA, ce se regăsește în driver-ul limbajului Pascal sau C++. Pentru a generaliza însă, vom considera modul menționat anterior, ce poate reda 16 culori, reprezentate pe 4 biți.

Fiecare culoare de bază are atribuită o constantă de la 0 la 15, precum urmează: 0 – **black** (negru); 1 – **blue** (albastru); 2 – **green** (verde); 3 – **cyan** (turcoaz); 4 – **red** (roșu); 5 – **magenta** (violet); 6 – **brown** (maro); 7 – **lightgrey** (gri deschis); 8 – **darkgrey** (gri închis); 9 – **lightblue** (albastru deschis); 10 – **lightgreen** (verde deschis); 11 – **lightcyan** (turcoaz deschis); 12 – **lightred** (roșu deschis); 13 – **lightmagenta** (violet deschis); 14 – **yellow** (galben) și 15 – **white** (alb).

! Aceste culori sunt cele implicate. Pentru a utiliza mai multe culori (dar nu în același timp), se poate schimba setul (**paleta**) de culori. Întrucât în acest moment nu sunt necesare o multitudine de culori, nu vom prezenta în detaliu acest aspect.

→ Pentru a seta **culoarea de fundal**, se utilizează procedura (în Pascal) sau funcția (în C++)

setbkcolor(culoare);

Exemplu: **setbkcolor(6);** sau **setbkcolor(RED);**

→ Selectarea **culorii cu care se desenează** se face cu ajutorul procedurii (în Pascal) sau funcției (în C++)

setcolor(culoare);

Exemplu: **setcolor(15);** sau **setcolor(WHITE);**

! Observații

- ✓ Schimbarea culorii nu afectează ce am desenat anterior, ci doar ce este scris după apelul acestei rutine.
- ✓ În cazul limbajului C++, numele simbolic al culorii se scrie obligatoriu cu majuscule.

Operația de desenare

Oricare ar fi modul de lucru ales, un punct se reprezintă printr-un pixel de coordonate **x** (linia) și **y** (coloana), ambele valori întregi. Punctul din stânga-sus are coordonatele (0,0). Pentru a ne muta la poziția (**x,y**), vom folosi procedura (în Pascal) sau funcția (în C++) **moveto(x,y)**. Pentru a trasa o linie de la punctul curent, determinat anterior, până la o nouă poziție, vom utiliza procedura (în Pascal) sau funcția (în C++) **lineto(x1,y1)**. Astfel, vom obține o linie între punctele de coordonate (**x,y**) și (**x1,y1**).

Exemplu. Mai jos, este prezentat un program ce desenează o linie pe diagonala principală a ecranului (de la colțul din stânga-sus la colțul din dreapta jos):

Varianța Pascal	Varianța C++
<pre>... begin initg; setcolor(red); moveto(0,0); lineto(getmaxx,getmaxy); readln; end.</pre>	<pre>... main() { init(); setcolor(RED); moveto(0,0); lineto(getmaxx(),getmaxy()); getch(); }</pre>

De asemenea, două funcții foarte utile sunt **getmaxx** și **getmaxy** (în Pascal) sau **getmaxx()** și **getmaxy()** (în C++). Acestea întorc valoarea minimă și respectiv, maximă a coordonatelor de pe ecran. Astfel, cu ajutorul lor se poate obține o independență relativă a programelor față de modul grafic al sistemului.

3.3.2. Curba lui Koch pentru un triunghi echilateral

Se consideră un triunghi echilateral. Fiecare latură a sa se transformă aşa cum se vede în figura următoare (se împarte în trei segmente congruente, se elimină segmentul din mijloc și se construiește deasupra un triunghi echilateral):



Figura 3.1. Exemplu de transformare

Fiecare latură a acestui poligon se transformă din nou, după aceeași regulă. Să se vizualizeze figura obținută după 1s pași (număr citit de la tastatură).

! Această curbă este cunoscută în literatura de specialitate ca fiind **curba lui Koch** (Herge von Koch a fost matematician suedez și a imaginat această curbă în anul 1904).

Programul principal va apela, pentru fiecare segment care constituie o latură a triunghiului, o procedură numită **generator**. Aceasta execută transformarea de 1s ori, având ca parametri de intrare coordonatele punctelor care constituie extremitățile segmentului, numărul de transformări făcute (n) și numărul de transformări care trebuie efectuate (1s). Pentru a înțelege funcționarea procedurii, trebuie să avem un minimum de cunoștințe specifice geometriei analitice (ce se poate face fără matematică?).

Fie AB un segment de dreaptă, unde A este un punct de coordonate (x_1, y_1) , iar B are coordonatele (x_2, y_2) . Două puncte P_1 și P_2 împart segmentul într-un anumit raport, notat cu k :

$$x_p = \frac{x_1 - k \cdot x_2}{1-k}, \quad y_p = \frac{y_1 - k \cdot y_2}{1-k}.$$

Demonstrați singuri aceste formule!

Fie segmentul AB cu A(x_1, y_1) și B(x_2, y_2). Considerăm punctele C și D care împart segmentul în trei segmente congruente.

Aflăm coordonata punctului D:

$$k = \frac{DA}{DB} = -2; \quad x_p = \frac{x_1 + 2 \cdot x_2}{3}, \quad y_p = \frac{y_1 + 2 \cdot y_2}{3}.$$

Problema constă în stabilirea coordonatelor vârfului noului triunghi echilateral. Acestea se obțin dacă se rotește punctul C în jurul punctului D cu unghiul $\pi/3$. Roatația o efectuează procedura **rotplan**.

Să prezentăm algoritmul care stă la baza procedurii **generator**:

- se pornește de la segmentul AB;
- se determină coordonatele punctului care constituie vârful triunghiului echilateral (să-l notăm cu V);
- în cazul în care segmentul nu a fost transformat de 1s ori, se apelează generator pentru segmentele AC, CV, VD și DB;
- contrar, se apelează procedura care trasează linia frântă ACVDB.

În programul principal au fost alese punctele care determină triunghiul echilateral initial – plasat în centrul ecranului – și pentru fiecare segment ce constituie o latură a acestuia s-a apelat procedura **generator**. Odată trasată curba, se colorează interiorul acesteia.

Programul este următorul:

Varianta Pascal	Varianta C++
<pre>uses graph,crt; var L,gdriver,gmode, 1s:integer; xmax,ymax:integer; procedure initg; ... procedure rotplan(xc,yc,x1, y1:integer; var x,y:integer; unghi:real); begin x := round(xc+(x1-xc)* cos(unghi)-(y1-yc)* sin(unghi)); y := round(yc+(x1-xc)* sin(unghi)+(y1-yc)* cos(unghi)); end; procedure desenez(x1,y1,x2, y2,x3,y3:integer); begin moveto(x1,y1); lineto((2*x1+x2)/3, (2*y1+y2)/3); lineto(x3,y3); lineto((x1+2*x2)/3, (y1+2*y2)/3); lineto(x2,y2); end; procedure generator(x1,y1,x2,y2, n,1s:integer); var x,y:integer; </pre>	<pre>#include "graphics.h" #include <iostream.h> #include <stdlib.h> #include <conio.h> #include <math.h> int gdriver,gmode,1s,L; void init() { ... void rotplan(int xc,int yc, int x1, int y1,int &x, int &y,float unghi) { x = ceil(xc+(x1-xc)*cos(unghi)- (y1-yc)*sin(unghi)); y = ceil(yc+(x1-xc)*sin(unghi)+ (y1-yc)*cos(unghi)); } void desenez(int x1,int y1, int x2,int y2,int x3,int y3) { moveto(x1,y1); lineto(div((2*x1+x2),3).quot, div((2*y1+y2),3).quot); lineto(x3,y3); lineto(div((x1+2*x2),3).quot, div((y1+2*y2),3).quot); lineto(x2,y2); } void generator(int x1,int y1, int x2,int y2,int n, int 1s) { int x,y; rotplan(div((2*x1+x2),3).quot, div((2*y1+y2),3).quot, div((x1+2*x2),3).quot, div((y1+2*y2),3).quot, x,y,M_PI/3); </pre>

```

begin
  rotplan((2*x1+x2) div 3,
  (2*y1+y2) div 3,(x1+2*x2) div
  3,(y1+2*y2) div 3,x,y,pi/3);
  if n<ls then
    begin
      generator(x1,y1,(2*x1+x2)
      div 3,(2*y1+y2) div 3,
      n+1,ls);
      generator((2*x1+x2) div 3,
      (2*y1+y2) div 3,
      x,y,n+1,ls);
      generator(x,y,(x1+2*x2) div
      3,(y1+2*y2) div 3,n+1,ls);
      generator((x1+2*x2) div 3,
      (y1+2*y2) div 3,
      x2,y2,n+1,ls);
    end
    else desenez(x1,y1,x2,y2,x,y);
  end;
begin
  write('ls= '); readln(ls);
  initg;
  setcolor(red);
  L:=getmaxx-320;
  generator(160,getmaxy()-150,
  160+L,getmaxy()-150,1,ls);
  generator(160+L,getmaxy()-
  150,160+div(L,2).quot,
  getmaxy()-150-
  ceil(L*(sqrt(3)/2)),1,ls);
  generator(160+div(L,2).quot,
  getmaxy()-150-
  ceil(L*(sqrt(3)/2)),160,
  getmaxy()-150,1,ls);
  setfillstyle(1,4);
  floodfill(div(getmaxx(),2)
  .quot,div(getmaxx(),
  2).quot,6);
  getch();
  closegraph();
end.
}

```

Prinți mai jos rezultatele obținute pentru diferite valori ale lui ls:

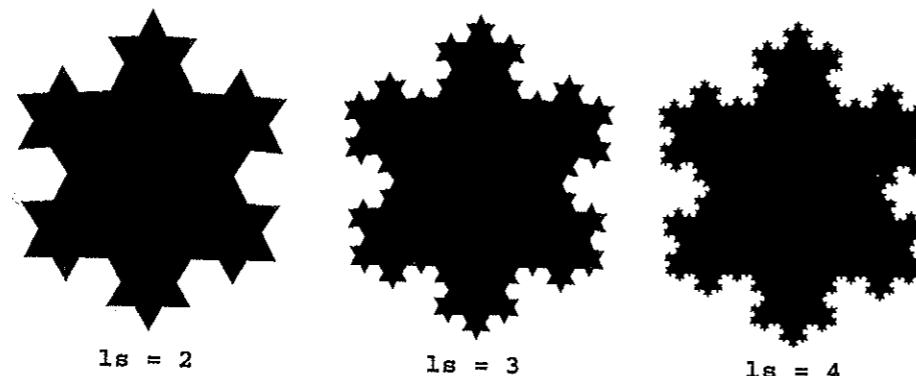


Figura 3.2. Exemple de fractali formați cu ajutorul curbei lui Koch (triunghi echilateral)

3.3.3. Curba lui Koch pentru un pătrat

Se consideră un pătrat. Fiecare latură a sa se transformă după cum se vede în figura de mai jos:

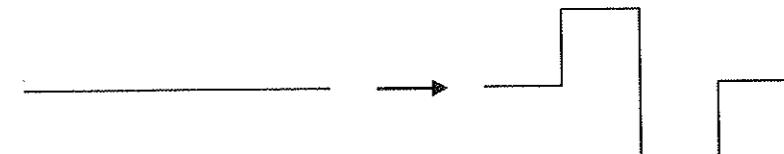


Figura 3.3. Exemplu de transformare

Fiecare segment al liniei frânte astfel formate se transformă din nou după aceeași regulă. Se cere să se vizualizeze curba după ls transformări (valoare citită de la tastatură). Transformarea și desenarea unui segment sunt realizate de procedura desen. Aceasta are ca parametri de intrare coordonatele punctului care determină segmentul, numărul de transformări efectuate (n) și numărul de transformări cerut (ls). Procedura conține următorul algoritm:

- dacă nu a fost efectuat numărul de transformări necesar, se calculează coordonatele punctelor care determină linia frântă obținută pornind de la segment și pentru fiecare segment din această linie se repeleză procedura desen;
- contrar, se desenează linia frântă obținută.

În final, figura se colorează. Programul este prezentat în continuare:

Varianta Pascal	Varianta C++
<pre> uses graph,crt; var gdriver,gmode,ls:integer; procedure initg; ... procedure rotplan(...); ... procedure desen(x1,y1,x2, y2,n,ls:integer); var x3,x4,x5,x6,x7,x8,xc, y3,y4,y5,y6,y7,y8,yc:integer; begin if n<=ls then begin x3:=(3*x1+x2) div 4; y3:=(3*y1+y2) div 4; rotplan(x3,y3,x1,y1,x4,y4, -pi/2); xc:=(x1+x2) div 2; end else begin generator(x1,y1,div((2*x1+x2), 3).quot,div((2*y1+y2), 3).quot,n+1,ls); generator(div((2*x1+x2), 3).quot,div((2*y1+y2), 3).quot,x,y,n+1,ls); generator(x,y,div((x1+2*x2), 3).quot,div((y1+2*y2), 3).quot,n+1,ls); generator(div((x1+2*x2), 3).quot,div((y1+2*y2), 3).quot,x2,y2,n+1,ls); end end end; ... procedure rotplan(x3,y3,x1,y1,x4,y4, -pi/2); begin generator(x1,y1,div((2*x1+x2), 3).quot,div((2*y1+y2), 3).quot,n+1,ls); generator(div((2*x1+x2), 3).quot,div((2*y1+y2), 3).quot,x,y,n+1,ls); generator(x,y,div((x1+2*x2), 3).quot,div((y1+2*y2), 3).quot,n+1,ls); generator(div((x1+2*x2), 3).quot,div((y1+2*y2), 3).quot,x2,y2,n+1,ls); end; end. } </pre>	<pre> #include "graphics.h" #include <iostream.h> #include <stdlib.h> #include <conio.h> #include <math.h> int gdriver,gmode,ls,L; void init() { ... } void rotplan(...) { ... } void desen(int x1,int y1,int x2,int y2,int n,int ls) { int x3,x4,x5,x6,x7,x8, xc,y3,y4,y5,y6,y7,y8,yc; if (n<=ls) { x3:=(3*x1+x2) div 4; y3:=(3*y1+y2) div 4; rotplan(x3,y3,x1,y1,x4,y4, -pi/2); xc:=(x1+x2) div 2; } else { generator(x1,y1,div((2*x1+x2), 3).quot,div((2*y1+y2), 3).quot,n+1,ls); generator(div((2*x1+x2), 3).quot,div((2*y1+y2), 3).quot,x,y,n+1,ls); generator(x,y,div((x1+2*x2), 3).quot,div((y1+2*y2), 3).quot,n+1,ls); generator(div((x1+2*x2), 3).quot,div((y1+2*y2), 3).quot,x2,y2,n+1,ls); } } </pre>

```

yc:=(y1+y2) div 2;
rotplan(xc,yc,x3,y3,x5,y5,
        -pi/2);
rotplan(xc,yc,x3,y3,
        x6,y6,pi/2);
x8:=(x1+3*x2) div 4;
y8:=(y1+3*y2) div 4;
rotplan(x8,y8,xc,yc,x7,y7,
        pi/2);
desen(x1,y1,x3,y3,n+1,ls);
desen(x3,y3,x4,y4,n+1,ls);
desen(x4,y4,x5,y5,n+1,ls);
desen(x5,y5,xc,yc,n+1,ls);
desen(xc,yc,x6,y6,n+1,ls);
desen(x6,y6,x7,y7,n+1,ls);
desen(x7,y7,x8,y8,n+1,ls);
desen(x8,y8,x2,y2,n+1,ls);
if n = 1s then begin
  moveto(x1,y1);
  lineto(x3,y3);
  lineto(x4,y4);
  lineto(x5,y5);
  lineto(x6,y6);
  lineto(x7,y7);
  lineto(x8,y8);
  lineto(x2,y2);
end
end;
begin
  write('1s= '); readln(ls);
  initg; setcolor(red);
  desen(100,100,300,100,1,ls);
  desen(300,100,300,300,1,ls);
  desen(300,300,100,300,1,ls);
  desen(100,300,100,100,1,ls);
  setfillstyle(1,blue);
  floodfill(div(getmaxx() div 2,
                getmaxy() div 2, red);
  readln
end.

```

```

xc=div(x1+x2,2).quot;
yc=div(y1+y2,2).quot;
rotplan(xc,yc,x3,y3,x5,y5,
        -M_PI/2);
rotplan(xc,yc,x3,y3,x6,y6,
        M_PI/2);
x8=div(x1+3*x2, 4).quot;
y8=div(y1+3*y2,4).quot;
rotplan(x8,y8,xc,yc,x7,y7,
        M_PI/2);
desen(x1,y1,x3,y3,n+1,ls);
desen(x3,y3,x4,y4,n+1,ls);
desen(x4,y4,x5,y5,n+1,ls);
desen(x5,y5,xc,yc,n+1,ls);
desen(xc,yc,x6,y6,n+1,ls);
desen(x6,y6,x7,y7,n+1,ls);
desen(x7,y7,x8,y8,n+1,ls);
desen(x8,y8,x2,y2,n+1,ls);
if (n == 1s)
  { moveto(x1,y1);
    lineto(x3,y3);
    lineto(x4,y4);
    lineto(x5,y5);
    lineto(x6,y6);
    lineto(x7,y7);
    lineto(x8,y8);
    lineto(x2,y2); }
}
main()
{
  cout<<"1s= "; cin>>ls;
  init(); setcolor(6);
  desen(100,100,300,100,1,ls);
  desen(300,100,300,300,1,ls);
  desen(300,300,100,300,1,ls);
  desen(100,300,100,100,1,ls);
  setfillstyle(1,3);
  floodfill(div(getmaxx(),2
               .quot,div(getmaxy(),2).quot,6);
  getch(); closegraph();
}

```

Sunt prezentate mai jos imaginile obținute în urma rulării programului, pentru diferite valori ale lui `1s`:

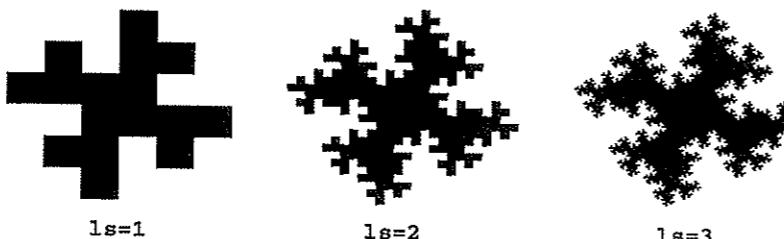


Figura 3.4. Exemple de fractali formați cu ajutorul curbei lui Koch (pătrat)

3.3.4. Arborele

Se dă un segment AB. Cu ajutorul lui se construiește un arbore, așa cum se vede în figura de mai jos:

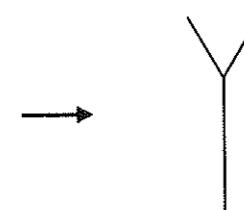


Figura 3.5. Exemplu de transformare în cazul unui arbore

Lungimea fiecărei ramuri este o treime din lungimea inițială a segmentului. Fiecare latură se transformă în mod asemănător. Se cere să se vizualizeze figura astfel rezultată, după `1s` transformări.

Pentru obținerea ramurilor se procedează astfel:

- se consideră punctul situat pe dreapta determinată de segment și pentru care avem:

$$k = \frac{CA}{CB} = 3; \quad x_c = \frac{x_1 - 3 \cdot x_2}{1-3} = \frac{3 \cdot x_2 - x_1}{2}, \quad y_p = \frac{y_1 - 3 \cdot y_2}{1-3} = \frac{3 \cdot y_2 - y_1}{2}.$$

- se rotește acest punct în jurul punctului B(x_2, y_2) cu un unghi de $\pi/4$;
- se rotește punctul în jurul lui B cu unghiul $-\pi/4$.

În urma acestor rotații se obțin coordonatele punctelor care, împreună cu punctul B, determină segmentele ce constituie ramurile arborelui.

Procedura `desenez` are ca parametri de intrare coordonatele unui segment, numărul de transformări efectuate (`n`) și numărul de transformări care trebuie efectuate (`1s`). În cazul în care nu s-au efectuat toate transformările, se trasează segmentul (cu o culoare oarecare), se calculează coordonatele punctelor care determină ramurile și, pentru fiecare segment, se reapeleză procedura.

Programul este prezentat mai jos:

Varianta Pascal	Varianta C++
<pre> uses graph,crt; var gdriver,gmode,ls:integer; xmax,ymax:integer; procedure initg; ... </pre>	<pre> #include "graphics.h" #include <iostream.h> #include <stdlib.h> #include <cconio.h> #include <math.h> int gdriver,gmode,ls,L; </pre>

```

procedure rotplan(xc,yc,x1,
y1:integer; var x,y:integer;
unghi:real);
...
procedure desenez(x1,y1,x2,y2,
n,ls:integer);
var x,y:integer;
begin
  if n<=ls then
    begin
      setcolor(1+random(15));
      moveto(x1,y1);
      lineto(x2,y2);
      rotplan(x2,y2,(3*x2-x1) div
      2,(3*y2-y1) div 2,x,y,pi/4);
      desenez(x2,y2,x,y,n+1,ls);
      rotplan(x2,y2,(3*x2-x1) div
      2,(3*y2-y1) div 2,x,y,-pi/4);
      desenez(x2,y2,x,y,n+1,ls);
    end
  end;
begin
  randomize;
  write('ls= '); readln(ls);
  initg;
  setbkcolor(white);
  desenez(getmaxx div 2,
  getmaxy, getmaxx div 2,
  getmaxy-250,1,ls);
  readln
end.

```

```

void init()
{ ... }

void rotplan(...)
{ ... }

void desenez(int x1,int y1,
int x2,int y2,int n,int ls)
{ int x,y;
  if (n<=ls)
  { setcolor(1+random(15));
    moveto(x1,y1);
    lineto(x2,y2);
    rotplan(x2,y2,div(3*x2-
    x1,2).quot,div(3*y2-y1,2)
    .quot,x,y,M_PI/4);
    desenez(x2,y2,x,y,n+1,ls);
    rotplan(x2,y2,div(3*x2-
    x1,2).quot,div(3*y2-y1,
    2).quot,x,y,-M_PI/4);
    desenez(x2,y2,x,y,n+1,ls);
  }
}

main()
{ randomize();
  cout<<"ls= "; cin>>ls;
  init(); setcolor(6);
  desenez(div(getmaxx(),2)
  .quot,getmaxy(),
  div(getmaxx(),2).quot,
  getmaxy()-250,1,ls);
  getch();
  closegraph();
}

```

Pentru diverse valori ale parametrului de intrare **ls**, vom obține arborii:

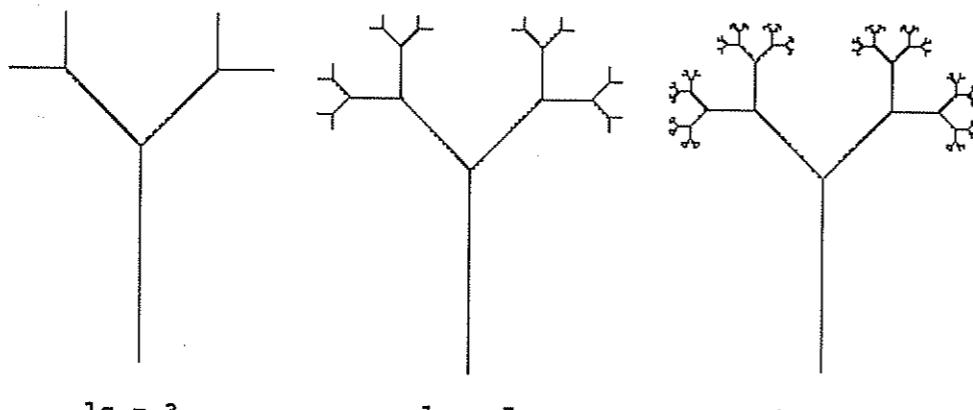


Figura 3.6. Exemple de fractali de tip arbore

Observații

- ✓ Exemplile grafice prezentate au fost generate pentru valori mici ale lui **ls** deoarece la tipărire, detaliile sunt greu de observat peste o anumită limită.
- ✓ Generarea fractalilor reprezintă o aplicație a recursivității, tehnica aplicată fiind **DIVIDE ET IMPERA**. Pentru valori mari ale lui **ls**, timpul de efectuare al calculelor poate fi și de ordinul zecilor de secunde, ceea ce poate fi considerat un inconvenient major.

Probleme propuse

1. Se citește $a \geq 1$, număr real. Se cere să se scrie o funcție care calculează $\ln(a)$ cu 3 zecimale exacte. Nu este permisă utilizarea funcției logaritmice a limbajului.
2. Scrieți o funcție care calculează prin metoda **DIVIDE ET IMPERA** suma numerelor reținute dintr-un vector.
3. Referitor la problema anterioară: care este complexitatea algoritmului folosit? Se va considera ca operație de bază adunarea.
4. Se citește un număr real $x \in (-10000, 10000)$. Să se afișeze partea fracționară. **Exemplu:** pentru $x=1.23$, se va afișa: 0.23; pentru $x=-12.7$, se va afișa 0.7. Nu se vor folosi funcții specializate ale limbajului.
5. Se știe că ecuația $x^3+x-1=0$ are o singură rădăcină reală în intervalul $(0,1)$. Scrieți un program, care o afișează cu 4 zecimale exacte.
6. **Problema selecției.** Se consideră un vector cu n componente numere naturale și $1 \leq t \leq n$. Se cere să se determine al t -lea cel mai mic element. Imaginea rezolvare care utilizează funcția **Poz** de la sortarea rapidă!
7. Se consideră un vector care reține n numere naturale. Se cere să se determine dacă există un element majoritar (adică un număr care se găsește în mai mult de $[n/2]+1$ elemente).

Victor Mitrana

8. Fiind dat x real, să se calculeze $\lfloor \sqrt[3]{x} \rfloor$ cu patru zecimale exacte! Nu se vor folosi funcții specializate ale limbajului.
9. Se pleacă de la un pătrat a cărui suprafață se divide în 9 părți egale prin împărțirea fiecărei laturi în 3 părți egale. Pătratul din mijloc se elimină. Cu pătratele rămase se procedează la fel. Vizualizați figura după **ls** astfel de transformări (*Covorul lui Sierpinski*).

Răspunsuri

1. $\ln(a)=x \Leftrightarrow a=e^x \Leftrightarrow e^x-a=0$. Dacă notăm cu $f(x)=e^x-a$, atunci trebuie rezolvată ecuația $f(x)=0$. Avem $f(0)=e^0-a=1-a<0$ și $f(a)=e^a-a>0$. De aici, rezultă că $f(x)$ are o rădăcină în intervalul $(0, a)$. Cum $f(x)$ este strict crescătoare (ca diferență între funcția strict crescătoare e^x și o constantă), rădăcina este unică. Algoritmul pe care îl folosim se numește în matematică "metoda înjumătățirii intervalului", dar, din punct de vedere informatic, corespunde metodei DIVIDE ET IMPERA.

Fie $li=0$ și $ls=a$, $m=(a+b)/2$. Dacă $f(li) \times f(m) < 0$, rădăcina se găsește în (li, m) , altfel rădăcina este în $[m, ls]$. Condiția de terminare este ca $|li-ls| < 0.0001$, pentru că trebuie să avem 3 zecimale exacte.

Varianta Pascal	Varianta C++
<pre>var a:real; function LogN(a,li,ls:double): double; begin if a=1 then LogN:=0 else if abs(li-ls)<0.0001 then LogN:=(li+ls)/2 else if (exp(li)-a)* (exp((li+ls)/2)-a)<0 then LogN:=LogN(a,li,(li+ls)/2) else LogN:=LogN(a,(li+ls)/2,ls) end; begin write ('a='); readln(a); writeln(' rezultat calculat:',LogN(a,0,a):3:3); writeln(' rezultat preluat ', ln(a):3:3); end.</pre>	<pre>#include <iostream.h> #include <math.h> double a; double LogN(double a,double li, double ls) { if (a==1) return 0; else if (fabs(li-ls)<0.0001) return (li+ls)/2; else if ((exp(li)-a)* (exp((li+ls)/2)-a)<0) return LogN(a,li, (li+ls)/2); else return LogN(a, (li+ls)/2,ls); } main() { cout<<"a="; cin>>a; cout<<"rezultat calculat " <<LogN(a,0,a)<<endl; cout<<"rezultat preluat " <<log(a)<<endl; }</pre>

Practic, la fiecare pas se înjumătăște intervalul în care se caută soluția și aceasta corespunde strategiei generale DIVIDE ET IMPERA.

2. Programul este prezentat mai jos:

Varianta Pascal	Varianta C++
<pre>type vector=array[1..9] of integer; var v:vector; n,i:integer;</pre>	<pre>#include <iostream.h> int n,i,v[10];</pre>

```
function
  Suma(li,ls:integer):integer;
begin
  if li=ls then Suma:=v[li]
  else Suma:=Suma(li,(li+ls) div
    2) + Suma((li+ls) div 2+1,ls);
end;

begin
  write('n='); readln(n);
  for i:=1 to n do readln(v[i]);
  writeln(suma(1,n));
end.
```

```
int Suma(int li, int ls)
{ if (li==ls) return v[li];
  else return
    Suma(li, (li+ls)/2)+ Suma((li+ls)/2+1,ls);
}

main()
{ cout<<"n="; cin>>n;
  for (i=1;i<=n;i++)
    cin>>v[i];
  cout<<Suma(1,n);
}
```

3. Fiecare problemă se descompune în alte două și rezultatul se adună. Pentru simplitate, considerați $n=2^k$. În final, se obține $O(n)$. Puteți scrie și funcția recursivă care calculează $T(n)$, dar, pentru a obține rezultatul corect, luați $n=2^k$:

$$T(n) = \begin{cases} 0 & n=1; \\ 2T\left(\frac{n}{2}\right) + 1 & \text{altfel.} \end{cases}$$

4. A calculă $[x]$ se reduce la DIVIDE ET IMPERA. Partea fracționară se obține ușor, dacă calculăm $|x-[x]|$. 5. Vedeți problema 1.

6. Funcția **Poz** returnează poziția k pe care se va găsi, după rularea ei, primul element al vectorului. În plus, toate elementele de indice mai mic decât k sunt mai mici sau egale decât $A[k]$ și toate elementele de indice mai mare decât k sunt mai mari sau egale decât $A[k]$. Altfel spus: elementul $A[1]$, care se află după rularea funcției pe poziția k , este al k -lea cel mai mic element din vectorul A . Atunci, în cazul în care $k=t$, problema este rezolvată. Dacă $t < k$, elementul căutat are indicele cuprins între li și $k-1$ și reluăm rularea funcției **Poz** între aceste limite, iar dacă $t > k$, elementul căutat are indicele între $k+1$ și ls și reluăm rularea funcției **Poz** între aceste limite. Datorită faptului că, la fiecare pas, se restrâng numărul valorilor de căutare, se ajunge în situația în care $t=k$. Secvența este:

Varianta Pascal	Varianta C++
<pre>li:=1; ls:=n; repeat poz(li,ls,k,a); if t<k then ls:=k-1; if t>k then li:=k+1; until t=k; writeln('Elementul cautat ', a[t]);</pre>	<pre>li=1; ls=n; do { poz(li,ls,k,a); if (t<k) ls=k-1; if (t>k) li=k+1; }while (t!=k); cout<<"Elementul cautat " <<a[t];</pre>

7. După aplicarea algoritmului de la problema anterioară, elementul din mijloc trebuie să fie majoritar.

Capitolul 4

Metoda BACKTRACKING

4.1. Prezentarea metodei

4.1.1. Când se utilizează metoda backtracking ?

Metoda backtracking se folosește în rezolvarea problemelor care îndeplinesc simultan următoarele condiții:

- soluția lor poate fi pusă sub forma unui vector $S = x_1, x_2, \dots, x_n$, cu $x_1 \in A_1, x_2 \in A_2, \dots, x_n \in A_n$;
- mulțimile A_1, A_2, \dots, A_n sunt mulțimi finite, iar elementele lor se consideră că se află într-o relație de ordine bine stabilită;
- nu se dispune de o altă metodă de rezolvare, mai rapidă.



În continuare, este prezentat un exemplu de problemă care poate fi rezolvat prin utilizarea tehnicii backtracking.

Generarea permutărilor. Se citește un număr natural n . Să se genereze toate permutările mulțimii $\{1, 2, \dots, n\}$. De exemplu, pentru $n=3$, permutările mulțimii $\{1, 2, 3\}$ sunt prezentate alăturat.

Pentru această problemă, $A_1=A_2=A_3=\{1, 2, 3\}$. Fie permutarea 213. Ea este scrisă sub formă de vector, unde $2 \in A_1, 1 \in A_2$ și $3 \in A_3$.

123
132
213
231
312
321

4.1.2. Principiul care stă la baza metodei backtracking

Principiul care stă la baza metodei backtracking va fi prezentat printr-un exemplu, acela al **generării permutărilor**. Cum se poate rezolva această problemă?

O primă soluție ar fi să generăm toate elementele produsului cartezian:

$$\begin{aligned} \{1, 2, 3\} \times \{1, 2, 3\} \times \{1, 2, 3\} &= \{11, 12, 13, 21, 22, 23, 31, 32, 33\} \\ \times \{1, 2, 3\} &= \{111, 112, 113, 121, 122, 123, 131, 132, 133, 211, \\ 212, 213, 221, 222, 223, 231, 232, 233, 311, 312, 313, 321, \\ 322, 323, 331, 332, 333\}. \end{aligned}$$

Apoi, urmează să vedem care dintre elementele acestui produs cartezian sunt permutări, adică să conțină numai numere distincte. Astfel, 111, 112... nu sunt permutări, dar 123 este permutare, ș.a.m.d. Produsul cartezian are 27 de elemente și dintre ele, doar 6 sunt permutări. În general, produsul cartezian are n^n elemente, din care permutări sunt doar $n!$. Vă dați seama că un astfel de algoritm de generare a permutărilor este ineficient...

Întrebarea este dacă problema nu se poate rezolva eficient? Să observăm că nu are rost să generăm un element al produsului cartezian pentru ca apoi, să ne dăm seama că nu este permutare, deoarece nu este alcătuit din numere distincte. De exemplu, dacă la un pas al algoritmului am generat 22, e clar că nu se poate obține o permutare, oricare ar fi numărul care urmează pe poziția 3.

Principiul metodei

- Metoda backtracking are la bază un principiu simplu: dacă în procesul de generare a unui vector soluție $S = x_1, x_2, \dots, x_n$, pentru componenta k , atunci când am generat deja x_1, x_2, \dots, x_k , constatăm că valoarea x_k nu este bine aleasă (păstrând-o nu se va ajunge la o soluție), nu trecem componenta $k+1$ ci reluăm căutarea pentru altă valoare pentru componenta k , iar dacă această valoare nu există, reluăm căutarea pentru componenta $k-1$.

Observați faptul că după ce am analizat posibilele valori pe care le poate lua componenta k , avem două posibilități: ori trecem la componenta $k+1$ (facem pasul **înainte**), ori mergem la componenta $k-1$ (facem pasul **înapoi**).

Trecem la exemplificarea algoritmului pentru generarea permutărilor, în cazul în care $n=3$.

- Componenta 1 va memora numărul 1. Întrucât există permutări care încep cu 1, trecem la elementul 2 - facem pasul **înainte**.

1		
---	--	--
- Componenta 2 va memora numărul 1.

1	1	
---	---	--
- Nu există permutări care încep cu 1,1, motiv pentru care, pentru aceeași componentă, vom reține valoarea următoare, adică 2. Întrucât există permutări care încep cu 1,2, vom trece la elementul 3 (**înainte**).

1	2	
---	---	--
- Componenta 3 va memora numărul 1.

1	2	1
---	---	---
- Nu există permutări care sunt de forma 1,2,1, motiv pentru care aceeași componentă va reține numărul următor, 2.

1	2	2
---	---	---
- Nu există permutări care sunt de forma 1,2,2, motiv pentru care aceeași componentă va memora numărul următor, adică 3. Am obținut deja o primă soluție și o afișăm.

1	2	3
---	---	---

- Pentru componenta 3, nu există o altă valoare pe care o putem utiliza. Din acest motiv, vom trece la elementul 2,

1	3	0
---	---	---

 (**înapoi**). Componenta 2 are deja memorată valoarea 2. Alegem valoarea următoare, 3. Întrucât există permutări care încep cu 1,3, vom trece la elementul următor, 3 (**înainte**).
- Prima valoare care poate fi memorată este 1. Întrucât nu există permutări de forma 1,3,1 trecem la valoarea următoare 2. Dar 1,3,2 este soluție și o afișăm.
- ...
- Algoritmul continuă până când se ajunge la componenta de indice 0. În acel moment, au fost deja afișate toate permutările.



Exercițiu. Arătați cum funcționează algoritmul până se ajunge la componenta de indice 0.

4.1.3. O modalitate de implementare a metodei backtracking

Pentru ușurarea înțelegерii metodei, mai întâi vom prezenta un subprogram general, aplicabil oricărei probleme. Subprogramul va apela alte subprograme care au întotdeauna același nume și parametri și care, **din punct de vedere al metodei**, realizează același lucru. Sarcina celui care face programul este să scrie explicit, pentru fiecare problemă în parte, subprogramele apelate de acesta.

- ✓ Evident, o astfel de abordare conduce la programe cu multe instrucțiuni. Din acest motiv, după înțelegerea metodei backtracking, vom renunța la această formă standardizată. Dar, principiul rămâne nemodificat.

Înălță subprogramul care implementează metoda. El va fi apelat prin **back(1)**.

Varianta Pascal	Varianta C++
<pre>procedure back(k:integer); begin if solutie(k) then tipar else begin init(k); while successor(k) do if valid(k) then back(k+1); end end; end;</pre>	<pre>void back(int k) { if (solutie(k)) tipar(); else { init(k); while(successor(k)) if (valid(k)) back(k+1); } }</pre>

Să-l analizăm! Subprogramul are parametrul **k** de tip întreg. Acest parametru are semnificația de indice al componentei vectorului pentru care se caută o valoare convenabilă. Algoritmul va porni cu componenta de indice 1. Din acest motiv, subprogramul se va apela cu **back(1)**. După cum observați, subprogramul este recursiv.

- ➔ Inițial se testează dacă s-a generat o soluție. Pentru aceasta, se apelează subprogramul **solutie(k)**.

Pentru permutări, vom avea o soluție când s-a generat o secvență alcătuită din **n** numere distincte. Cum subprogramul este recursiv, acest fapt se întâmplă atunci când s-a ajuns pe nivelul **n+1**.

- ➔ Dacă s-a obținut o soluție, aceasta se afișează. Pentru această operație se va utiliza subprogramul **tipar**.
- ➔ În situația în care nu a fost obținută o soluție, se inițializează nivelul **k**. Inițializarea se face cu valoarea aflată înaintea tuturor valorilor posibile. Se va folosi subprogramul **init**.

Pentru permutări, inițializarea se face cu 0.

- ➔ După inițializare, se generează, pe rând, **toate** valorile mulțimii **A_k**. Pentru aceasta se utilizează subprogramul **successor**. Rolul său este de a atribui componentei **k** valoarea următoare celei deja existente.
- ➔ Pentru fiecare valoare generată, se testează dacă aceasta îndeplinește anumite **condiții de continuare**. Acest test este realizat de subprogramul **valid**:

- ➔ În cazul în care **condițiile sunt îndeplinite**, se trece la componenta **k+1**, urmând ca generația valorilor pe nivelul **k** să continue atunci când se revine pe acest nivel;
- ➔ dacă **condițiile de continuare** nu sunt îndeplinite, se generează următoarea valoare pentru componentă **k**.
- ➔ După ce au fost generate toate valorile mulțimii **A_k** se trece, implicit, la componenta **k-1**, iar algoritmul se încheie când **k=0**.

Pentru permutări, pe fiecare nivel, valorile posibile sunt **A_k={1,2,...,n}**. Condiția de continuare, în acest caz este ca valoarea aflată pe nivelul **k** să fie distinctă în raport cu valorile aflate pe nivelurile inferioare.

Programul de generare a permutărilor este prezentat în continuare.

Varianta Pascal	Varianta C++
<pre> var n:integer; sol:array [1..10] of integer; procedure init(k:integer); begin sol[k]:=0 end; function succesor (k:integer):boolean; begin if sol[k]<n then begin sol[k]:=sol[k]+1; succesor:=true end else succesor:=false end; function valid (k:integer):boolean; var i:integer; begin valid:=true; for i:=1 to k-1 do if sol[i]=sol[k] then valid:=false end; function solutie (k:integer):boolean; begin solutie:=(k=n+1) end; procedure tipar; var i:integer; begin for i:=1 to n do write(sol[i]); writeln end; procedure back(k:integer); begin if solutie(k) then tipar else begin init(k); while succesor(k) do if valid(k) then back(k+1) end end; begin write('n='); readln(n); back(1) end. </pre>	<pre> #include <iostream.h> #include <iostream.h> int n, sol[10]; void init(int k) { sol[k]=0; } int succesor(int k) { if (sol[k]<n) { sol[k]++; return 1; } else return 0; } int valid(int k) { int i, ev=1; for (i=1;i<=k-1;i++) if (sol[k]==sol[i]) ev=0; return ev; } int solutie(int k) { return k==n+1; } void tipar() { for (int i=1;i<=n;i++) cout<<sol[i]; cout<<endl; } void back(int k) { if (solutie(k)) tipar(); else { init(k); while(succesor(k)) if (valid(k)) back(k+1); } } main() { cout<<"n="; cin>>n; back(1); } </pre>

4.1.4. Problema celor n dame



□ **Enunț.** Fiind dată o tablă de șah cu dimensiunea $n \times n$, se cer toate soluțiile de aranjare a n dame, astfel încât să nu se afle două dame pe aceeași linie, coloană sau diagonală (damele să nu se atace reciproc).

De exemplu, dacă $n=4$, o soluție este reprezentată în figura 4.1., a). Modul de obținere al soluției este prezentat în figurile următoare, de la b) la i):

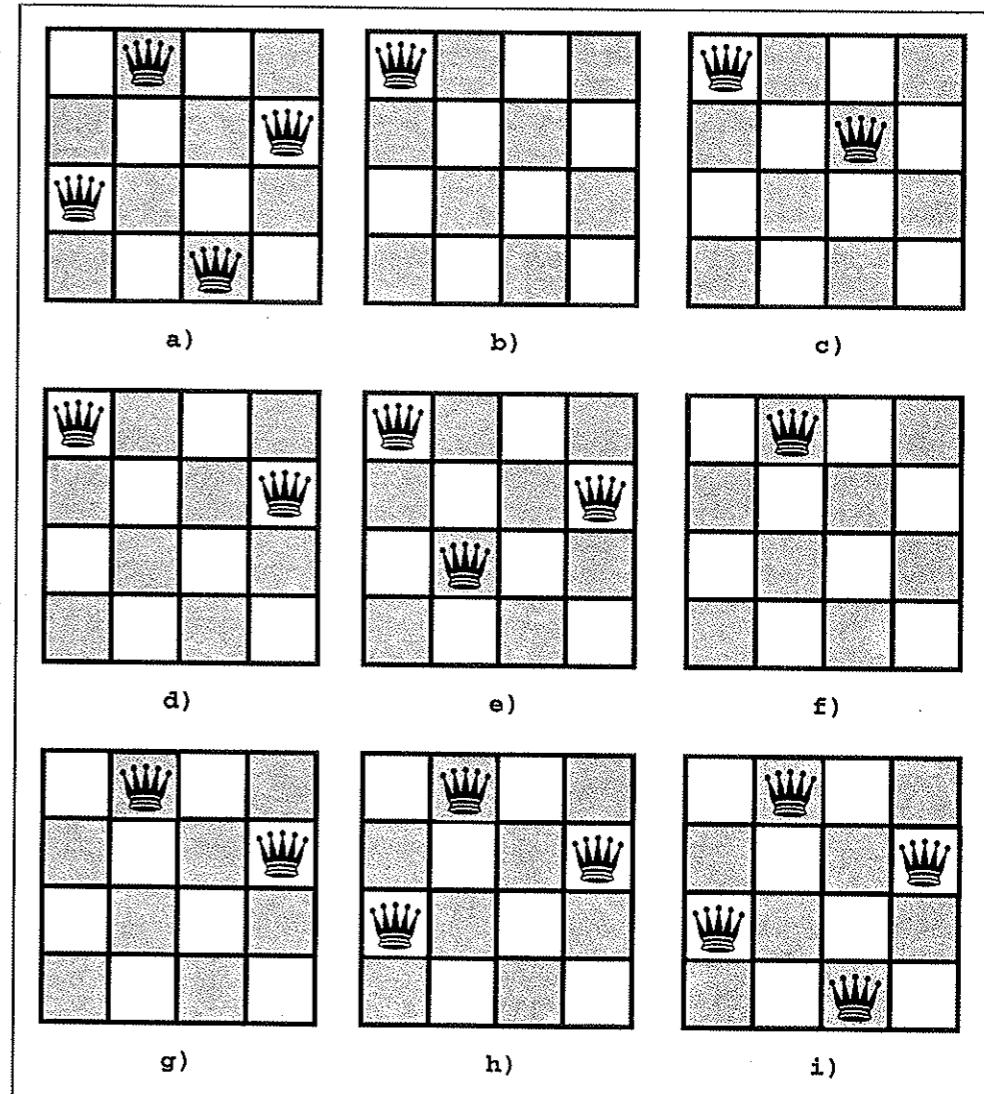


Figura 4.1. Exemplu pentru $n=4$

Comentarii referitoare la figurile anterioare

- b) Observăm că o damă trebuie să fie plasată singură pe linie. Poziționăm prima damă pe linia 1, coloana 1.
- c) A doua damă nu poate fi așezată decât în coloana 3.
- d) Observăm că a treia damă nu poate fi plasată în linia 3. Încercăm atunci plasarea celei de-a doua dame în coloana 4.
- e) A treia damă nu poate fi plasată decât în coloana 2.
- f) În această situație dama a patra nu mai poate fi așezată. Încercând să avansăm cu dama a treia, observăm că nu este posibil să o plasăm nici în coloana a-3-a, nici în coloana a-4-a, deci o vom scoate de pe tablă. Dama a doua nu mai poate avansa, deci și ea este scoasă de pe tablă. Avansăm cu prima damă în coloana a-2-a.
- g) A doua damă nu poate fi așezată decât în coloana a 4-a.
- h) Dama a treia se așează în prima coloană.
- i) Acum este posibil să plasăm a patra damă în coloana 3 și astfel am obținut o soluție a problemei.

Algoritmul continuă în acest mod până când trebuie scoasă de pe tablă prima damă.

- Pentru căutarea și reprezentarea unei soluții folosim un vector cu n componente, numit **sol** (având în vedere că pe fiecare linie se găsește o singură damă). Prin **sol[i]** înțelegem coloana în care se găsește dama de pe linia i .

Alăturat, puteți observa modul în care este reprezentată soluția cu ajutorul vectorului **sol**.

2	4	1	3
---	---	---	---

- Două dame se găsesc pe aceeași diagonală dacă și numai dacă este îndeplinită condiția:

$$|sol(i)-sol(j)|=|i-j|$$

(diferența, în modul, dintre linii și coloane este aceeași).

Exemple:

a)

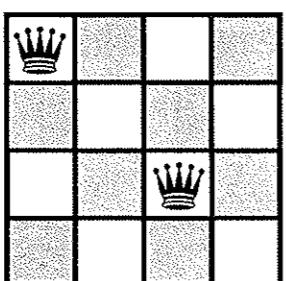


Figura 4.2.

$$\begin{aligned} sol(1) &= 1 \quad i = 1 \\ sol(3) &= 3 \quad j = 3 \\ |sol(1) - sol(3)| &= |1 - 3| = 2 \\ |i - j| &= |1 - 3| = 2 \end{aligned}$$

b)

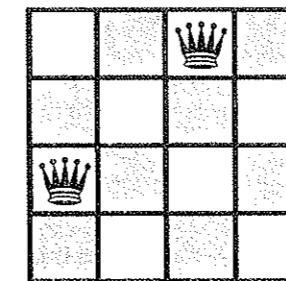


Figura 4.3.

$$\begin{aligned} sol(1) &= 3 \quad i = 1 \\ sol(3) &= 1 \quad j = 3 \\ |sol(i) - sol(j)| &= |3 - 1| = 2 \\ |i - j| &= |1 - 3| = 2 \end{aligned}$$

Întrucât două dame nu se pot găsi în aceeași coloană, rezultă că o soluție este sub formă de permutare. O primă idee ne conduce la generarea tuturor permutărilor și la extragerea soluțiilor pentru problemă (ca două dame să nu fie plasate în aceeași diagonală). Dacă procedăm astfel, înseamnă că nu lucrăm conform strategiei backtracking. Aceasta presupune ca imediat ce am găsit două dame care se atacă, să reluăm căutarea în alte condiții. Față de programul de generare a permutărilor, programul de generare a tuturor soluțiilor problemei celor n dame are o singură condiție suplimentară, în subprogramul **valid**. Mai jos, puteți observa noua versiune a subprogramului **valid**. Dacă îl utilizați în locul subprogramului cu același nume din programul de generare a permutărilor, veți obține programul care rezolvă problema celor n dame.

Varianta Pascal	Varianta C++
<pre>function valid(k:integer): boolean; var i:integer; begin valid:=true; for i:=1 to k-1 do if (sol[k]=sol[i]) or (abs(sol[k]-sol[i])=abs(k-i)) then valid:=false end;</pre>	<pre>int valid(int k) { for (int i=1;i<k;i++) if (sol[k]==sol[i] abs(sol[k]-sol[i])==abs(k-i)) return 0; return 1; }</pre>

Problema este un exemplu folosit în mai toate lucrările în care este prezentată metoda backtracking.



În ceea ce ne privește, dincolo de un exemplu de backtracking, am avut ocazia să vedem căt de mult seamănă rezolvările (prin această metodă) a două probleme între care, aparent, nu există nici o legătură.

**Exerciții**

1. Desenați configurația tablei corespunzătoare vectorului **sol=(3,1,4,2,5)** și verificați dacă aceasta reprezintă o soluție a problemei damelor.

2. Explicați de ce configurația corespunzătoare vecorului `sol=(3, 1, 3, 4, 5)` nu este o soluție a problemei damelor. Care este poziția din `sol` la care nu s-a făcut alegerea corectă a unei valori valide?

3. Explicați de ce nu orice permutare a mulțimii $\{1, 2, \dots, n\}$ este o soluție a problemei damelor pe o tablă cu n linii și n coloane.

4. Determinați, folosind metoda backtracking, o soluție care se obține pe o tablă cu șase linii și șase coloane, știind că dama plasată pe ultima linie trebuie să se afle pe a doua coloană. Considerați că mai este util să completăm vectorul `sol` pornind de la poziția 1? Dacă pornim de la poziția 1, ce adaptări trebuie făcute?

4.2. Mai puține linii în programul sursă

Până în prezent, am rezolvat două probleme prin metoda backtracking: generarea permutărilor și problema celor n dame. În ambele cazuri, am utilizat **subprogramul standard**.

Este întotdeauna necesar să-l folosim pe acesta sau putem reține numai ideea și, după caz, să scriem mai puțin?

Evident, după ce am înțeles bine metoda, putem renunța la subprogramul standard. Pentru aceasta, **încorporăm în subprogramul standard unele dintre subprogramele pe care le-ar fi apelat**.

Vom exemplifica această încorporare pentru problema celor n dame, deja rezolvată standardizat.

- Putem elimina subprogramul `init`. Este foarte ușor de realizat această operație. În locul apelului vom scrie instrucțiunea prin care componentei `k` i se atribuie 0.
- Putem elimina subprogramul `solutie`. În locul apelului vom testa dacă `k` este egal cu `n+1`.
- Putem elimina subprogramul `tipar`. În locul apelului vom scrie secvența prin care se afișează `st`.
- Putem elimina subprogramul `succesor`. În locul apelului vom testa dacă `st[k] < n` și avem grija să incrementăm valoarea aflată pe nivelul curent.

Puteți observa în continuare programul obținut. Este cu mult mai scurt! Oricum, ideea de rezolvare rămâne aceeași. Subprogramele prezentate au fost numai încorporate, nu s-a renunțat la ele.

Varianța Pascal	Varianța C++
<pre> var sol:array[1..9] of integer; n:integer; function valid(k:integer):boolean; var i:integer; begin valid:=true; for i:=1 to k-1 do if (sol[k]=sol[i]) or (abs(sol[k]-sol[i])=abs(k-i)) then valid:=false end; procedure back(k:integer); var i:integer; begin if k=n+1 {solutie} then begin {tipar} for i:=1 to n do write(sol[i]); writeln; end else begin sol[k]:=0; {init} while sol[k]<n do {succesor} begin sol[k]:=sol[k]+1; if valid(k) then back(k+1) end end end; begin write('n='); readln(n); back(1); end. </pre>	<pre> #include <iostream.h> #include <math.h> int n, sol[10]; int valid(int k) { for (int i=1;i<k;i++) if (sol[k]==sol[i] abs(sol[k]-sol[i]) ==abs(k-i)) return 0; return 1; } void back(int k) { if (k==n+1) // solutie //tipar { for (int i=1;i<=n;i++) cout<<sol[i]; cout<<endl; } else { sol[k]=0; while(sol[k]<n) // succesor { sol[k]++; if (valid(k))back(k+1); } } } main() { cout<<"n="; cin>>n; back(1); } </pre>

Uneori veți întâlni și o rezolvare precum următoarea, care în subprogramul `back`, pentru `succesor` se folosește o instrucțiune repetitivă de tip `for`:

Varianta Pascal	Varianta C++
<pre> procedure back(k:integer); var i:integer; begin if k=n+1 then begin for i:=1 to n do write(sol[i]); writeln; end else for i:=1 to n do begin sol[k]:=i; if valid(k) then back(k+1) end end; </pre>	<pre> void back(int k) { int i; if (k==n+1) { for (i=1;i<=n;i++) cout<<sol[i]; cout<<endl; } else for (i=1;i<=n;i++) { sol[k]=i; if (valid(k)) back(k+1); } } </pre>



Exerciții

- Testați subprogramul anterior pentru problema generării permutărilor.
- Adaptați rezolvarea problemei permutărilor astfel încât să se afișeze numai permutările în care oricare două numere consecutive **nu sunt** alăturate.
- Observați că ordinea de afișare a soluțiilor depinde de ordinea în care se consideră elementele mulțimilor **A₁, A₂, ...**. Ce modificări trebuie aduse procedurii recursive **back** astfel încât permutările de 4 elemente să fie afișate în ordinea: **4321, 4312, 4231, 4213, 4132, 4123, 3421, 3412 ... 1243, 1234?**
- Renunțați la utilizarea subprogramului **valid**, utilizând un vector **folosit**, în care **folosit[i]** are valoarea **0** dacă numărul **i** nu este deja folosit în soluție și are valoarea **1** în caz contrar. Astfel, plasarea valorii **i** în vectorul soluție (**sol[k]←i**) trebuie însoțită de memorarea faptului că **i** este utilizat (**folosit[i]←1**), la revenirea din recursie (cand se înlătură valoarea de pe poziția curentă) fiind necesară memorarea faptului că **i** nu mai este utilizat în soluție (**folosit[i]←0**). Condiția de validare se reduce în acest caz la:

Dacă **folosit[i]=0** atunci ...

- Urmăriți toate modalitățile diferite de a așeza patru obiecte identificate prin numerele **1, 2, 3, 4** pe un cerc, la distanțe egale. Vom observa că nu toate permutările de patru obiecte sunt configurații distincte, datorită distribuției pe cerc. Astfel permutările **1234, 2341, 3412 și 4123** reprezintă una și aceeași configurație. Scrieți un program care afișează numai permutările distincte conform așezării pe un cerc. **Indicație:** se va considera **sol[1]=1** și se vor permuta doar celelalte elemente.

4.3. Cazul în care se cere o singură soluție.

Exemplificare: problema colorării hărților

Sunt probleme care se rezolvă cu metoda backtracking și în care se cere o singură soluție.

Implementarea "ca la carte" presupune utilizarea unei variabile de semnalizare (de exemplu, variabila **gata**) care să fie inițial **0**, la obținerea soluției dorite aceasta primind valoarea **1**. Orice succesor va fi condiționat în plus de valoarea variabilei **gata**.

De această dată, pentru simplitate, vom renunța la programarea structurată și vom opri în mod forțat programul.

- ➔ În Pascal, veți utiliza procedura **halt**.
- ➔ În C++, veți folosi funcția **exit**, cu parametrul **EXIT_SUCCESS** (o constantă). Pentru a o putea utiliza, trebuie să includeți fișierul antet "**stdlib.h**".

```
#include<stdlib.h>
```

În ambele cazuri, secvența care determină oprirea forțată este trecută imediat după ce prima soluție a fost afișată.



Exercițiu. Modificați programul care rezolvă problema celor **n** dame, astfel încât acesta să afișeze o singură soluție.

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

! Este demonstrat faptul că sunt suficiente numai **4** culori pentru ca orice hartă să poată fi colorată.

Pentru exemplificare, vom considera harta din figura 4.4., unde țările sunt numerotate cu cifre cuprinse între **1** și **5**.

O soluție a acestei probleme este următoarea:

- țara 1 - culoarea **1**;
- țara 2 - culoarea **2**;
- țara 3 - culoarea **1**;
- țara 4 - culoarea **3**;
- țara 5 - culoarea **4**.

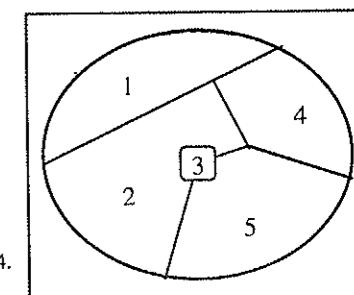


Figura 4.4.

Harta este furnizată programului cu ajutorul unei matrice (tablou) $A_{n,n}$:

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

Matricea **A** este simetrică. Pentru rezolvarea problemei se utilizează vectorul **sol**, unde **sol[k]** reține culoarea atașată țării **k**. Evident, orice soluție are exact **n** componente.

Varianta Pascal	Varianta C++
<pre> var sol:array[1..9] of integer; a:array[1..10,1..10] of integer; n,i,j:integer; function valid(k:integer):boolean; begin valid:=true; for i:=1 to k-1 do if (sol[k]=sol[i]) and (a[k,i]=1) then valid:=false end; procedure back(k:integer); var i:integer; begin if k=n+1 then begin for j:=1 to n do write(sol[j]); halt; end else for i:=1 to n do begin sol[k]:=i; if valid(k) then back(k+1); end; end; begin write('Numarul de tari='); readln(n); for i:=1 to n do for j:=1 to i-1 do begin write('a[,i,,',j,']='); readln(a[i,j]); a[j,i]:=a[i,j] end; back(1); end. </pre>	<pre> #include <iostream.h> #include <stdlib.h> int n,i,j,sol[10],a[10][10]; int valid(int k) { for (int i=1;i<k;i++) if (sol[k]==sol[i] && a[i][k]==1) return 0; return 1; } void back(int k) { int i; if (k==n+1) { for (int i=1;i<=n;i++) cout<<sol[i]; exit(EXIT_SUCCESS); } else for (i=1;i<=4;i++) { sol[k]=i; if (valid(k)) back(k+1); } } main() { cout<<"Numarul de tari="; cin>>n; for (int i=1;i<=n;i++) for (int j=1;j<=i-1;j++) { cout<<"a["<<i <<', '<<j<<"]="; cin>>a[i][j]; a[j][i]=a[i][j]; } back(1); } </pre>

Exerciții

- Soluția afișată este și soluția care utilizează un număr minim de culori?
- Dacă țările din centrul figurii alăturate sunt numerotate cu 1, 2, 3, 4, iar cele de la exterior cu 5 și 6, care este soluția afișată de programul dat? Este acesta numărul minim de culori necesare?
- Câte culori sunt suficiente pentru colorarea unei hărți particulare în care orice țară se învecinează cu cel mult două țări?
- Dați exemplu de particularitate pe care poate să o aibă o hartă pentru a fi suficiente două culori pentru colorarea tuturor țărilor?

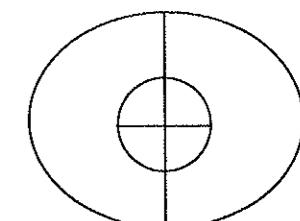


Figura 4.5.

4.4. Aplicații ale metodei backtracking în combinatorică

4.4.1. O generalizare utilă

Acum, că am învățat să generăm permutările mulțimii $\{1, 2, \dots, n\}$, se pune problema să vedem de ce este util acest algoritm. La ce folosește faptul că putem aranja numerele $\{1, 2, \dots, n\}$ în toate modurile posibile?

Să observăm că acest algoritm poate fi folosit pentru a aranja **oricare** **n** elemente distincte în toate modurile posibile.

Ex: Exemple

1. Se dă o mulțime alcătuită din **n** litere distincte. Se cer **toate** cuvintele care se pot forma cu ele, astfel încât fiecare cuvânt să conțină **n** litere distincte. De exemplu, dacă mulțimea este $\{a, b, c\}$, vom avea cuvintele: **abc**, **acb**, **bac**, **bca**, **cab** și **cba**.

2. Se dau numele a **n** persoane. Se cere să se afișeze **toate** modurile posibile în care acestea se pot așeza pe o bancă. De exemplu, dacă **n=3**, iar persoanele sunt **Ioana, Costel și Mihaela**, atunci soluțiile sunt:

Ioana Costel Mihaela;
Ioana Mihaela Costel;
Costel Ioana Mihaela;
...

În astfel de cazuri, cele n elemente distincte se memorează într-un vector v , aşa cum vedeti mai jos:

a	b	c
1	2	3

Ioana	Costel	Mihaela
1	2	3

Atunci când s-a generat o permutare, de exemplu 213, vom afișa $v[2]v[1]v[3]$, adică bac, în primul caz sau Costel Ioana Mihaela, în al doilea caz.

Procedeul de mai sus poate fi folosit pentru oricare altă aplicație din combinatorică, în probleme cum ar fi: generarea tuturor submulțimilor unei mulțimi, generarea aranjamentelor, a combinațiilor sau a tuturor părților unei mulțimi.



Exercițiu. Scrieți programul care rezolvă exemplul 2.



Mulțimea permutărilor mulțimii $\{1, 2, \dots, n\}$ reprezintă toate funcțiile bijective $f: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$. De exemplu, dacă $n=3$, permutarea 213 este funcția $f: \{1, 2, 3\} \rightarrow \{1, 2, 3\}$ definită astfel: $f(1)=2$; $f(2)=1$; $f(3)=3$.

4.4.2. Produs cartezian

Enunț. Se dau n mulțimi: A_1, A_2, \dots, A_n , unde $A_i = \{1, 2, \dots, k_i\}$, pentru $k=1, 2, \dots, n$. Se cere produsul cartezian al celor n mulțimi.

Exemplu: $A_1 = \{1, 2\}$, $A_2 = \{1, 2, 3\}$, $A_3 = \{1, 2, 3\}$.

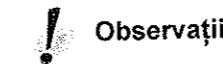
$A_1 \times A_2 \times A_3 = \{(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 2, 1), (1, 2, 2), (1, 2, 3), (1, 3, 1), (1, 3, 2), (1, 3, 3), (2, 1, 1), (2, 1, 2), (2, 1, 3), (2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 3, 1), (2, 3, 2), (2, 3, 3)\}$.

Rezolvare. De la început observăm că este necesar să afișăm **toate** soluțiile. Să observăm că o soluție este de forma x_1, x_2, \dots, x_n , cu $x_1 \in A_1$, $x_2 \in A_2$, ..., $x_n \in A_n$. De aici rezultă necesitatea folosirii unui vector sol , cu n componente, unde $sol[1]$ va conține numerele naturale între 1 și k_1 , $sol[2]$ va conține numerele naturale între 1 și k_2 , ..., $sol[n]$ va conține numerele naturale între 1 și k_n . Observăm că valorile care pot fi luate $sol[1]$ sunt între 1 și k_1 , valorile care pot fi luate $sol[2]$ sunt între 1 și k_2 , ..., valorile care pot fi luate $sol[n]$ sunt între 1 și k_n . Pentru a putea reține aceste valori, vom utiliza un vector a , cu n componente, unde $a[1] = k_1$, $a[2] = k_2$, ..., $a[n] = k_n$. Pentru exemplul dat, vectorul a va reține (2, 3, 3).

Important! Să observăm că orice valoare reținută de $sol[i]$ între 1 și k_i îndeplinește condițiile de continuare (este validă). Din acest motiv, nu mai este necesar să utilizăm subprogramul **valid**.

Mai jos, puteți observa programul care generează produsul cartezian al mulțimilor date:

Varianta Pascal	Varianta C++
<pre> var n,i:integer; sol,a:array[1..10] of integer; procedure back(k:integer); begin if k=n+1 then begin for i:=1 to n do write(sol[i]); writeln end else begin sol[k]:=0; while sol[k]<a[k] do begin sol[k]:=sol[k]+1; back(k+1) end end end; begin write('Numarul de multimi='); readln(n); for i:=1 to n do begin write('a[,i,]='); readln(a[i]) end; back(1) end. </pre>	<pre> #include <iostream.h> int n, sol[10], a[10], i; void back(int k) { if (k==n+1) { for (i=1;i<=n;i++) cout<<sol[i]; cout<<endl; } else { sol[k]=0; while (sol[k]<a[k]) { sol[k]++; back(k+1); } } } main() { cout<<"Numarul de multimi="; cin>>n; for(int i=1;i<=n;i++) { cout<<"a["<<i<<"]="; cin>>a[i]; } back(1); } </pre>



Observații

- Avem $k_1 \times k_2 \times \dots \times k_n$ elemente ale produsului cartezian. De aici rezultă că algoritmul este exponențial.
- O altă interpretare pentru metoda backtracking: fiind date n mulțimi: A_1, A_2, \dots, A_n , produsul cartezian al lor $A_1 \times A_2 \times \dots \times A_n$ se mai numește **spațiu soluțiilor**. În acest context, metoda backtracking caută una sau toate soluțiile, care sunt elemente ale produsului cartezian și care îndeplinesc anumite condiții. Astfel, se poate justifica faptul că, în generarea produsului cartezian, nu este necesar subprogramul **valid** pentru că se generează toate elementele produsului cartezian, fără a verifica anumite condiții.

Deși algoritmul este exponențial, există aplicații utile, evident, atunci când fiecare mulțime A_i poate lua numai câteva valori și unde n este suficient de mic.



Exerciții

1. Tabelarea anumitor funcții. Se dă funcția

$$f : A_1 \times A_2 \times \dots \times A_n \rightarrow \mathbb{R},$$

unde fiecare mulțime A_i este dată de numerele întregi din intervalul $[a_i, b_i]$ și

$$f = c_1 x_1 + c_2 x_2 + \dots + c_n x_n, c_i \in \mathbb{R}.$$

Se cere să se realizeze un tabel, în care pentru fiecare valoare din domeniul de definiție să se afișeze valoarea funcției corespunzătoare acelei valori.

2. Scrieți programul care generează toate "cuvintele" cu patru litere care au prima și ultima literă vocală, litera a doua consoană din mulțimea {P, R, S, T}, iar a treia literă consoană din mulțimea {B, M, R, T, V}.

3. Scrieți programul care generează și numără câte cuvinte de cinci litere ale alfabetului englez se pot forma, cu condiția să nu existe două consoane alăturate și nici două vocale alăturate.

4.4.3. Generarea tuturor submulțimilor unei mulțimi

Enunț. Fiind dată mulțimea $A=\{1, 2, \dots, n\}$, se cere să se afișeze toate submulțimile ei.

Rezolvare. Să ne amintim că submulțimile unei mulțimi A se pot reprezenta prin vectorul caracteristic v , unde:

$$V[i] = \begin{cases} 1, & \text{dacă } i \in A \\ 0, & \text{dacă } i \notin A \end{cases}$$

De exemplu, dacă $A=\{1, 2, 3\}$, pentru submulțimea $\{1, 3\}$ vom avea $V=(1, 0, 1)$. De aici, rezultă că problema se reduce la generarea tuturor valorilor posibile pe care le poate reține vectorul caracteristic.

Aceasta înseamnă că o soluție este de forma x_1, x_2, \dots, x_n , unde $x_i \in \{0, 1\}$. și în acest caz, orice valoare ar reține componenta i , ea nu trebuie să îndeplinească nici o condiție de continuare, motiv pentru care subprogramul **valid** nu este necesar.

În continuare, puteți observa programul care generează toate valorile pe care le poate reține vectorul caracteristic:

Varianta Pascal	Varianta C++
<pre> var n,i:integer; sol:array[1..10] of integer; procedure back (k:integer); begin if k=n+1 then begin for i:=1 to n do write(sol[i]); writeln; end else begin sol[k]:=-1; while sol[k]<1 do begin sol[k]:=sol[k]+1; back(k+1); end; end; begin write('n='); readln(n); back(1); end; end; begin write('n='); readln(n); back(1); end. </pre>	<pre> #include <iostream.h> int n, sol[10], i; void back(int k) { if (k==n+1) { for (i=1;i<=n;i++) cout<<sol[i]; cout<<endl; } else { sol[k]=-1; while (sol[k]<1) { sol[k]++; back(k+1); } } } main() { cout<<"n="; cin>>n; back(1); } </pre>



Exerciții

1. Problema nu este rezolvată în totalitate. Programul afișează numai toate valorile pe care le poate lua vectorul caracteristic. Completăți-l astfel încât programul să afișeze toate submulțimile mulțimii $\{1, 2, \dots, n\}$!

2. Se citesc numele a 6 elevi. Afiați toate submulțimile mulțimii celor 6 elevi.

3. Să se afișeze toate numerele scrise în baza 10 a căror reprezentare în baza 2 are n cifre, dintre care exact k sunt egale cu 1. Valorile n și k se citesc de la tastatură ($n < 12$, $k < n$). De exemplu, pentru $n=3$ și $k=2$, se obțin valorile: 5 și 6.

4. Realizați un program care generează combinații de n cifre 0 și 1 cu proprietatea că în orice grup de 3 cifre consecutive există cel puțin o cifră de 1. De exemplu, dacă $n=4$, se afișează combinațiile: 0010, 0011, 0100, 0101, 0110, 0111, 1001, 1010, 1011, 1100, 1101, 1110, 1111.

5. Se citesc două numere naturale n și s ($n < 10$, $s < 1000$). Să se afișeze mulțimile formate din n numere prime cu proprietatea că suma elementelor din fiecare mulțime este exact s .

Observații

- ✓ Fiind dată o mulțime cu n elemente, avem 2^n submulțimi ale ei. Mulțimea dată și submulțimea vidă sunt submulțimi ale mulțimii date! De ce? Fiecare componentă a vectorului characteristic poate reține două valori. Prin urmare, numărul de submulțimi este

$$\underbrace{2 \cdot 2 \cdot 2 \cdots 2}_{\text{de } n \text{ ori}} = 2^n.$$

De aici rezultă că algoritmul care generează toate submulțimile mulțimii $1, 2, \dots, n$ este exponential.

- ✓ Uneori, veți rezolva probleme în care se dă o mulțime și se cere o submulțime a sa care îndeplinește anumite caracteristici. În anumite situații, problema se poate rezolva prin utilizarea unor algoritmi mai rapizi (polinomiali). Greșeala tipică care se face este că se generează toate submulțimile, după care se selectează cea (cele) care îndeplinește condițiile date.

Exemplu. Se dă o mulțime de numere reale. Se cere să se determine o submulțime a sa care are suma maximă. Problema se rezolvă ușor: se consideră ca făcând parte din submulțime numai numerele pozitive. Altfel, dacă am genera toate submulțimile...

4.4.4. Generarea combinărilor

Fiind dată mulțimea $A=\{1, 2, \dots, n\}$, se cer toate submulțimile ei cu p elemente. Problema este cunoscută sub numele de "generarea combinărilor de n , luate câte p ". Se știe că numărul soluțiilor acestei probleme este

$$C_n^p = \frac{n!}{(n-p)!p!}.$$

De exemplu, dacă $n=4$ și $p=3$, soluțiile sunt următoarele:

$$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\} \text{ și } \{2, 3, 4\}.$$

Enunț. Se citesc n și p numere naturale, $n \geq p$. Se cere să se genereze toate submulțimile cu p elemente ale mulțimii $A=\{1, 2, \dots, n\}$.

Rezolvare. O soluție este de forma x_1, x_2, \dots, x_p , unde $x_1, x_2, \dots, x_p \in A$. În plus, x_1, x_2, \dots, x_p trebuie să fie distințe. Cum la o mulțime ordinea elementelor nu prezintă importanță, putem genera elementele ei în ordine strict crescătoare. Această observație ne ajută foarte mult în elaborarea algoritmului.

a) Pentru $k > 1$, $\text{sol}[k] > \text{sol}[k-1]$.

- b) Pentru fiecare $k \in \{1, 2, \dots, p\}$, $\text{sol}[k] \leq n-p+k$. Să presupunem, prin absurd, că această ultimă relație nu este respectată. Aceasta înseamnă că $\exists k$, astfel încât $\text{sol}[k] > n-p+k$. Deci:

$$\begin{aligned} \text{sol}[k+1] &> n-p+k+1, \\ \dots \\ \text{sol}[p] &> n-p+p=n. \end{aligned}$$

Absurd. De aici rezultă că:

$$\begin{aligned} 1 \leq \text{sol}[1] \leq n-p+1, \\ \text{sol}[1] < \text{sol}[2] \leq n-p+2, \\ \dots \\ \text{sol}[n-1] < \text{sol}[n] \leq n-p+p=n. \end{aligned}$$

Relațiile de mai sus simplifică mult algoritm, pentru că ținând cont de ele, nu mai este necesar să se testeze nici o condiție de continuare.

Varianta Pascal	Varianta C++
<pre> var sol:array[1..9] of integer; n,p:integer; procedure back(k:integer); var i:integer; begin if k=p+1 then begin for i:=1 to p do write(sol[i]); writeln; end else begin if k>1 then sol[k]:=sol[k-1] else sol[k]:=0; while sol[k]<n-p+k do begin sol[k]:=sol[k]+1; back(k+1); end end; end; end; begin write('n='); readln(n); write ('p='); readln(p); back(1); end. </pre>	<pre> #include <iostream.h> int n,p,sol[10]; void back(int k) { int i; if (k==p+1) { for (i=1;i<=p;i++) cout<<sol[i]; cout<<endl; } else { if (k>1) sol[k]=sol[k-1]; else sol[k]=0; while (sol[k]<n-p+k) { sol[k]++; back(k+1); } } } main() { cout<<"n="; cin>>n; cout<<"p="; cin>>p; back(1); } </pre>

 Examinând raționamentul propus putem observa că, în anumite cazuri, analiza unei probleme conduce la un algoritm cu mult mai rapid.



Exerciții

- Se dă coordonatele din plan a n puncte. Afipați coordonatele vârfurilor tuturor pătratelor care au ca vârfuri puncte din mulțimea considerată.
- Se dă n substanțe chimice. Se știe că, în anumite condiții, unele substanțe intră în reacții chimice cu altele. Fiind date p perechi de forma (i, j) cu semnificația că substanța i intră în reacție cu substanța j , se cer toate grupurile de $s < n$ substanțe astfel încât oricare două substanțe din grup nu intră în reacție.

4.4.5. Generarea aranjamentelor

Se dă două mulțimi $A = \{1, 2, \dots, p\}$ și $B = \{1, 2, \dots, n\}$. Se cer toate funcțiile injective definite pe A cu valori în B . O astfel de problemă este una de generare a aranjamentelor de n luate câte p (A_n^p).

Exemplu: $p=2$, $n=3$. Avem: 12, 21, 13, 31, 23, 32. De exemplu, 21 este funcția $f: A \rightarrow B$ dată astfel: $f(1)=2$; $f(2)=1$. Avem relațiiile:

$$A_n^p = \frac{n!}{(n-p)!} = n(n-1)\dots(n-p+1).$$

□ **Enunț.** Se citesc n și p . Să se genereze toate aranjamentele de n luate câte p .

Să observăm că dacă se cunoaște fiecare submulțime de p elemente a mulțimii de n elemente, atunci aranjamentele se pot obține permutând în toate modurile posibile elementele unei astfel de mulțimi. Pornind de la această observație, suntem tentați să generăm toate submulțimile cu p elemente ale mulțimii cu n elemente și, din fiecare astfel de submulțime, să obținem permutările ei. **Exercițiu!**

Pe de altă parte, se poate lucra mult mai eficient. O soluție este de formă: $x_1 x_2 \dots x_p$, unde $x_1, x_2, \dots, x_p \in B$. În plus, x_1, x_2, \dots, x_p trebuie să fie distințe. Spre deosebire de algoritmul de generare a combinărilor, aici ne interesează toate permutările unei soluții (acestea sunt, la rândul lor, alte soluții). Aceasta înseamnă că nu mai putem pune în soluție elementele în ordine crescătoare. Să recapitulăm:

- o soluție are p numere din B ;
- numerele trebuie să fie distințe.

Rezultă de aici că algoritmul este același de la permutări, diferența fiind dată de faptul că soluția are p numere, nu n ca în cazul permutărilor.

Varianta Pascal	Varianta C++
<pre> var sol:array[1..9]of integer; n,p:integer; function valid(k:integer):boolean; var i:integer; begin valid:=true; for i:=1 to k-1 do if sol[k]=sol[i] then valid:=false end; procedure back(k:integer); var i,j:integer; begin if k=p+1 then begin for j:=1 to p do write(sol[j]); writeln end else for i:=1 to n do begin sol[k]:=i; if valid(k) then back(k+1) end end; begin readln(n); readln(p); back(1) end. </pre>	<pre> #include <iostream.h> int n,p,sol[10]; int valid(int k) { for (int i=1;i<k;i++) if (sol[k]==sol[i]) return 0; return 1; } void back(int k) { int i,j; if (k==p+1) { for (j=1;j<=p;j++) cout<<sol[j]; cout<<endl; } else for (i=1;i<=n;i++) { sol[k]=i; if (valid(k)) back(k+1); } } main() { cin>>n; cin>>p; back(1); } </pre>



Exerciții

- Se citesc n , p și apoi n litere distințe. Afipați toate cuvintele care se pot forma cu p dintr-o ele.
- Se citesc n și apoi numele mici a n persoane. Știind că toate numele care se termină cu a reprezintă nume de fată, celelalte fiind nume de băieți, să se afișeze toate mulțimile de perechi fată-băiat care se pot forma. Două mulțimi sunt distințe dacă cel puțin una dintre perechi diferă. De exemplu, pentru $n=5$, Maria, Ana, Doina, Doru, Cosmin, se afișează mulțimile: {Maria-Doru, Ana-Cosmin}, {Ana-Cosmin, Maria-Doru}, {Maria-Doru, Doina-Cosmin}, {Doina-Doru, Maria-Cosmin}, {Ana-Doru, Doina-Cosmin}, {Doina-Doru, Ana-Cosmin}.

3. Cei n acționari ai unei firme trebuie să organizeze un număr maxim de ședințe tip masă rotundă la care să participe exact p dintre ei. Știind că oricare două ședințe trebuie să difere fie prin acționarii prezenți, fie prin vecinii pe care îi au aceștia la masă, stabiliți numărul de ședințe pe care le pot organiza. De exemplu, dacă $n=4$ și $p=3$, atunci sunt posibile 5 configurații diferite ale celor 3 acționari așezăți la masa rotundă: 1-2-3; 1-3-2; 1-3-4; 1-4-3; 2-3-4; 2-4-3 (configurațiile 2-3-1 și 3-1-2 nu se consideră, deoarece sunt echivalente, la masa rotundă, cu configurația 1-2-3).

4.4.6. Generarea tuturor partițiilor mulțimii $\{1, 2, \dots, n\}$

 **Definiția 4.1.** Fie mulțimea $A=\{1, 2, \dots, n\}$. Se numește **partiție** a mulțimii A , un set de $k \leq n$ mulțimi care îndeplinesc condițiile de mai jos:

- a) $A_1 \cup A_2 \cup \dots \cup A_k = A$;
- b) $A_i \cap A_j = \emptyset, \forall i \neq j \in \{1, 2, \dots, n\}$.

Exemplu. Considerăm mulțimea $A=\{1, 2, 3\}$. Avem partițiile:

$\{1, 2, 3\}$
 $\{1, 2\} \{3\}$
 $\{1, 3\} \{2\}$
 $\{2, 3\} \{1\}$
 $\{1\} \{2\} \{3\}$

Enunț. Se citește un număr natural, n . Se cer toate partițiile mulțimii $A=\{1, 2, \dots, n\}$.

Rezolvare. Chiar dacă știm să generăm toate submulțimile unei mulțimi, tot nu ne ajută să generăm toate partițiile.

1. Pentru a putea genera toate partițiile, trebuie să găsim o metodă prin care să putem reține o partiție. O primă idee ne conduce la folosirea unui vector, sol , astfel: dacă $sol[i]=k$, atunci elementul i se găsește în mulțimea k a partiției. Totuși, nu știm câte mulțimi sunt în partiția respectivă. Există o partiție care conține n mulțimi atunci când fiecare element este într-o mulțime și una care conține toate mulțimile, adică tocmai mulțimea A . Cu alte cuvinte, numărul mulțimilor dintr-o partiție este între 1 și n .

2. Pentru a avea o ordine în generarea soluțiilor, elementele mulțimii A trebuie să aparțină de submulțimi consecutive ale partiției.

→ Din acest motiv, $sol[i]$ va lua valori între 1 și

$$1 + \max\{sol[1], sol[2], \dots, sol[i-1]\}.$$

Prin această condiție se evită situația în care, de exemplu, vectorul sol reține $(1, 3, 1)$. Aceasta ar avea semnificația că elementele 1 și 3 se găsesc în submulțimea 1 a partiției, iar elementul 2 se găsește în submulțimea 3 a partiției. În acest caz, lipsește submulțimea 2 a partiției.

Să exemplificăm funcționarea algoritmului pentru cazul $n=3$:

- $sol=(1, 1, 1)$ - $A_1=\{1, 2, 3\}$;
- $sol=(1, 1, 2)$ - $A_1=\{1, 2\} A_2=\{3\}$;
- $sol=(1, 2, 1)$ - $A_1=\{1, 3\} A_2=\{2\}$;
- $sol=(1, 2, 2)$ - $A_1=\{1\} A_2=\{2, 3\}$;
- $sol=(1, 2, 3)$ - $A_1=\{1\} A_2=\{2\} A_3=\{3\}$.

! Să observăm că nici în cazul acestei probleme nu trebuie să verificăm existența anumitor condiții de continuare.

Analizați programul astfel obținut!

Varianta Pascal	Varianta C++
<pre> var sol:array[0..10]of integer; n,i,j,maxim:integer; procedure tipar; begin maxim:=1; for i:=2 to n do if maxim<sol[i] then maxim:=sol[i]; writeln('Partitie '); for i:=1 to maxim do begin for j:=1 to n do if sol[j]=i then write (j,' '); writeln; end; end; procedure back (k:integer); var i,j,maxprec:integer; begin if k=n+1 then tipar else begin maxprec:=0; for j=1 to k-1 do if maxprec<sol[j] then maxprec:=sol[j]; for (i=1; i<=maxprec+1; i++) if sol[k]=i then max[k]:=sol[k]; back(k+1); end; end; </pre>	<pre> #include <iostream.h> int n, sol[10], max[10], i, j, maxim; void tipar() { maxim=1; for (i=2; i<=n; i++) if (maxim<sol[i]) maxim=sol[i]; cout<<"Partitie "<<endl; for (i=1; i<=maxim; i++) { for (j=1; j<=n; j++) if (sol[j]==i) cout<<j<< " "; cout<<endl; } } void back(int k) { int i, j, maxprec; if (k==n+1) tipar(); else { maxprec=0; for (j=1; j<=k-1; j++) if (maxprec<sol[j]) maxprec=sol[j]; for (i=1; i<=maxprec+1; i++) if (sol[k]==i) max[k]=sol[k]; back(k+1); } } </pre>

```

for i:=1 to maxprec+1 do
begin
  sol[k]:=i;
  back(k+1)
end;
end;

begin
  write('n='); readln(n);
  back(1);
end.

```

```

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

```

Exercițiu. Puteți arăta că oricărei partiții îi aparține un unic conținut al vectorului **sol**, obținut ca în program?

Indicație. Observați că întotdeauna elementul 1 aparține primei submulțimi a partiției, elementul 2 poate apartine submulțimilor 1 sau 2 ale partiției, ..., elementul **n** poate apartine submulțimilor, 1, 2 sau **n** ale partiției. Pornind de aici, construiți vectorul **sol**!

Tinând cont de faptul că oricărei partiții îi corespunde un unic conținut al vectorului **sol** și oricărui conținut al vectorului **sol** îi corespunde o unică partiție, am obținut, practic, o funcție bijectivă de la mulțimea partițiilor mulțimii **A** la mulțimea conținuturilor generate de algoritm ale vectorului **sol**. Pornind de la această bijecție, în loc ca algoritmul să genereze partiții, el va determina conținuturile vectorului **sol**. Apoi, pentru fiecare conținut al vectorului **sol**, se obține o partiție.

Exercițiu. Modificați programul precedent pentru ca acesta să afișeze toate partiții care conțin exact 3 submulțimi.

4.5. Alte tipuri de probleme care se rezolvă prin utilizarea metodei backtracking

4.5.1. Generalități

Toate problemele pe care le-am întâlnit până acum admit soluții care îndeplinesc următoarele caracteristici:

- soluțiile sunt sub formă de vector;
- toate soluțiile unei probleme au aceeași **lungime**, unde prin lungime înțelegem numărul de componente ale vectorului soluție.

Exemple. Fie mulțimea **A={1, 2...n}**. Atunci:

- Toate permutările mulțimii **A** au lungimea **n**.
- Toate submulțimile cu **p** elemente ale mulțimii **A** (generarea combinărilor) au lungimea **p**.
- Toate soluțiile sub formă de vector ale problemei generării tuturor partiților mulțimii **A** au lungimea **n**.

În realitate, cu ajutorul metodei backtracking se pot rezolva și probleme care nu îndeplinesc condițiile de mai sus. Astfel, există probleme în care nu se cunoaște de la început lungimea soluției, există probleme care admit mai multe soluții de lungimi diferite, există probleme în care soluția este sub formă unei matrice cu două sau trei linii etc. Exemplile următoare vă vor convinge.

4.5.2. Generarea partiților unui număr natural

Enunț. Se citește un număr natural **n**. Se cere să se tipărească toate modurile de descompunere a lui ca sumă de numere naturale. De exemplu, pentru **n=4**, avem: **4, 31, 22, 211, 13, 121, 112, 1111**.

Ordinea numerelor din sumă este importantă. Astfel, se tipărește **112** dar și **211, 121**.

Rezolvare. De la început, observăm că nu se cunoaște lungimea unei soluții. Ea poate fi cuprinsă între 1, în cazul în care numărul în sine constituie o descompunere a sa și **n**, atunci când numărul este descompus ca sumă a **n** numere egale cu 1.

Trecem la stabilirea **algoritmului** pe care îl vom folosi.

- Fiecare componentă a vectorului **sol** trebuie să rețină o valoare mai mare sau egală cu 1.
- Mai întâi să observăm că, în procesul de generare a soluțiilor, trebuie ca în permanență să fie respectată relația

$$\text{sol[1]} + \text{sol[2]} + \dots + \text{sol[k]} \leq n.$$

- Avem soluție atunci când

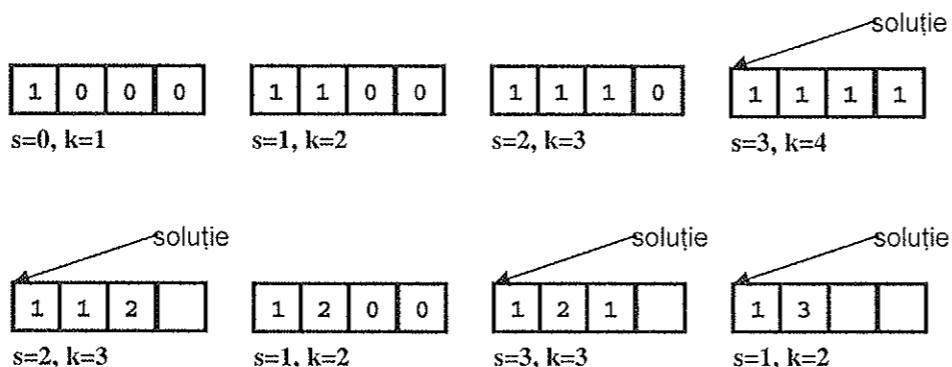
$$\text{sol[1]} + \text{sol[2]} + \dots + \text{sol[k]} = n.$$

Rezultă de aici că trebuie să cunoaștem, la fiecare pas **k**, suma

$$s = \text{sol[1]} + \text{sol[2]} + \dots + \text{sol[k-1]}.$$

O primă posibilitate ar fi ca la fiecare pas să calculăm această sumă. Dar, se poate lucra eficient. Suma va fi reținută în permanență într-o variabilă globală, numită **s**.

Mai jos, este prezentată funcționarea algoritmului pentru **n=4**:



! Observați modul în care calculăm suma la fiecare pas. De câte ori se trece la componenta următoare (**k+1**), la **s** se adună **sol[k]**, de câte ori se face pasul înapoi (se trece la componenta **k-1**), din **s** se scade **sol[k]**.

Programul este prezentat în continuare:

Varianta Pascal	Varianta C++
<pre>var sol:array[1..100] of integer; n,i,s:integer; procedure back (k:integer); begin if s=n then begin for i:=1 to k-1 do write(sol[i]); writeln; end else begin sol[k]:=0; while sol[k]+s<n do begin sol[k]:=sol[k]+1; s:=s+sol[k]; back(k+1); s:=s-sol[k] end; end; begin write('n='); readln(n); back(1); end.</pre>	<pre>#include <iostream.h> int sol[100], n,i,s; void back(int k) { if (s==n) { for (i=1;i<=k-1;i++) cout<<sol[i]; cout<<endl; } else { sol[k]=0; while (sol[k]+s<n) { sol[k]++; s+=sol[k]; back(k+1); s-=sol[k]; } } } main() { cout<<"n="; cin>>n; back(1); }</pre>

Exerciții

1. Cum trebuie procedat în cazul în care se cere ca soluțiile să fie afișate o singură dată? Spre exemplu, dacă s-a afișat descompunerea **1,1,2** să nu se mai afișeze **2,1,1** sau **1,2,1**?

Indicație: procedeul a mai fost întâlnit, de exemplu la generarea combinațiilor. Soluțiile se vor genera în ordine crescătoare. Modificați programul în acest sens.

2. Adaptați metoda de rezolvare astfel încât să se genereze numai partitiiile formate din numere naturale distințe.

3. Adaptați metoda de rezolvare astfel încât să se genereze numai partitiiile formate din cel puțin **p** numere naturale distințe (**n** și **p** citite de la tastatură).

4. Adaptați metoda de rezolvare astfel încât să se genereze numai partitiiile formate din numere naturale aflate în intervalul **[a,b]** (**n, a** și **b** citite de la tastatură).

5. Rezolvați problema scrierii numărului natural **n** ca sumă de numere naturale alese dintr-o mulțime formată din **k** valori date **{v1, v2, ..., vk}**. Astfel, **10** se poate scrie ca sumă de numere alese din mulțimea **{2,3,6}** în felul următor:

$$10=2+2+2+2+2, \quad 10=2+2+3+3, \quad 10=2+2+6.$$

4.5.3. Plata unei sume cu bancnote de valori date

Enunț. Se dă suma **s** și **n** tipuri de monede având valori de **a₁, a₂, ..., a_n** lei. Se cer toate modalitățile de plată a sumei **s** utilizând aceste monede. Se presupune că se dispune de un număr nelimitat de exemplare din fiecare tip de monedă.

lată soluțiile pentru **Suma=5, n=3** (trei tipuri de monede) cu valorile **1, 2, 3**:

- 1) 1 de 2, 1 de 3; 2) 1 de 1, 2 de 2; 3) 2 de 1, 1 de 3;
4) 3 de 1, 1 de 2; 5) 5 de 1;

Rezolvare. Valorile celor **n** monede sunt reținute de vectorul **a**. Astfel, **a[1]** va reține valoarea monedei de tipul 1, **a[2]** valoarea monedei de tipul 2, și.a.m.d. Numărul de monede din fiecare tip va fi reținut de vectorul **sol**. Astfel, **sol[1]** va reține numărul de monede de tipul 1, **sol[2]** va reține numărul de monede de tipul 2, și.a.m.d. În aceste condiții, o soluție pentru exemplul anterior arată ca alăturat, unde suma 5 se formează cu două monede de 1 și o monedă de 3.

! Ce observăm?

sol	2	0	1
a	1	2	3

1. Există componente ale vectorului **sol** care rețin 0. Această situație corespunde cazului în care moneda respectivă nu este luate în calcul. Din acest motiv, fiecare componentă a vectorului **sol** va fi inițializată cu o valoare aflată înaintea tuturor celor posibile, adică cu -1.

2. Orice soluție are exact **n** componente (**n** este numărul de tipuri de monede). Acest număr include toate monedele, chiar și cele care nu sunt luate în calcul.

3. Ca și la problema anterioară, vom reține în permanentă suma obținută la un moment dat. Astfel, la pasul **k**, avem la dispoziție suma:

$$s = a[1]*sol[1] + a[2]*sol[2] + \dots + a[k-1]*sol[k-1]$$

4. Avem soluție dacă:

$$s = a[1]*sol[1] + a[2]*sol[2] + \dots + a[n]*sol[n] = \text{Suma}$$

 **Exercițiu.** Încercați să arătați, prin desene, ca la problema anterioară, modul de obținere a tuturor soluțiilor pentru **Suma=5** și **n=3**.

În continuare, este prezentat programul:

Varianța Pascal	Varianța C++
<pre>var sol,a:array[1..100] of integer; n,i,s,Suma:integer; procedure back (k:integer); begin if s==suma then begin writeln('Solutie'); for i:=1 to k-1 do if sol[i]>>0 then writeln (sol[i], ' monede de ',a[i]); writeln; end else begin sol[k]:=-1; while (sol[k]*a[k]+s<Suma) && k<n+1) { sol[k]++; s+=sol[k]*a[k]; back(k+1); s-=sol[k]*a[k]; } end; end;</pre>	<pre>#include <iostream.h> int sol[100], a[100], n,i,s,Suma; void back(int k) { if (s==Suma) { cout<<"Solutie "<<endl; for(i=1;i<=k-1;i++) if(sol[i]) cout<<sol[i]<<"monede de " <<a[i]<<endl; cout<<endl; } else { sol[k]=-1; while(sol[k]*a[k]+s<Suma && k<n+1) { sol[k]++; s+=sol[k]*a[k]; back(k+1); s-=sol[k]*a[k]; } } } main() { cout<<"suma="; cin>>Suma; cout<<"n="; cin>>n;</pre>

```
begin
  write ('suma=');
  readln(suma);
  write('n='); readln(n);
  for i:=1 to n do
    begin
      write ('a[,i,]=');
      readln(a[i]);
    end;
  back(1);
}
```



Exercițiu. Adaptați rezolvarea problemei plății unei sume cu bancnote date, cunoscând, în plus, pentru fiecare valoare **a_i** numărul limită **b_i** de bancnote cu valoarea respectivă disponibile. Astfel, pentru **s=100**, **a=(2,5,50)**, **b=(10,6,3)**, varianta **s=10x5+1x50** nu corespunde cerinței deoarece nu avem la dispoziție 10 monede de 5, ci doar 6.

4.5.4. Problema labirintului

Enunț. Se dă un labirint sub formă de matrice cu **m** linii și **n** coloane. Fiecare element al matricei reprezintă o cameră a labirintului. Într-o din camere, de coordonate **lin** și **col**, se găsește un om. Se cere să se găsească toate ieșirile din labirint. Nu este permis ca un drum să treacă de două ori prin aceeași cameră.

O primă problemă care se pune este precizarea modului de codificare a ieșirilor din fiecare cameră a labirintului.

Fie **1(i,j)** un element al matricei. Acesta poate lua valori între 0 și 15. Se consideră ieșirile spre nord, est, sud și vest, luate în această ordine. Pentru fiecare direcție cu ieșire se reține 1, iar în caz contrar, se reține 0. Un sir de patru cifre 1 sau 0 formează un număr în baza 2. Acest număr este convertit în baza 10 și reținut în **1(i,j)**. De exemplu, pentru o cameră care are ieșire în nord și vest, avem $1001_{(2)}=9_{(10)}$.

Exemplu. Alăturat este prezentat un labirint. Acolo unde nu este permisă trecerea dintr-o cameră în alta, se marchează cu o linie oblică. De asemenea, matricea reține și valorile corespunzătoare ieșirilor, așa cum sunt ele cerute de program.

15	11	10	14
11	12	11	12
11	6	15	7

✓ Rezolvare

1. O cameră vizitată se reține prin coordonatele ei: **lin** (linia) și **col** (colana). Din acest motiv, pentru a reține un traseu vom utiliza o matrice cu două coloane și mai multe linii: **sol**. De exemplu, dacă camera initială este cea de coordonate (2,2) o soluție este (2,2), (2,3), (1,3).

2. Nu toate soluțiile au aceeași lungime, întrucât există trasee de lungime diferită. Se obține o soluție atunci când coordonatele camerei unde s-a intrat sunt în afara matricei (nu au linia între 1 și m și nu au coloana între 1 și n). Evident, atunci când s-a găsit o situație, aceasta se afișează.

3. Spunem că o cameră este accesibilă dacă există intrare din camera curentă către ea. Atenție la modul (vedeți programul) în care testăm dacă o cameră este accesibilă sau nu. Este o operație în care se testează conținutul unui anumit bit. Acesta se obține efectuând un **ȘI** logic între două valori. De exemplu, dacă testăm ieșirea spre sud, atunci efectuăm **ȘI** logic între $0010_{(2)} = 2_{(10)}$ și valoarea reținută în matrice pentru camera curentă. Dacă valoarea obținută este diferită de 0, atunci avem ieșire din camera curentă către sud.

4. Înainte de a intra într-o cameră accesibilă se testează dacă respectiva cameră a mai fost vizitată sau nu. Pentru aceasta utilizăm funcția **vizitat**. În caz că a fost vizitată, se face pasul înapoi.

Analizați programul:

Varianta Pascal	Varianta C++
<pre>var sol:array [1..100,1..2] of integer; l:array [0..10,0..10] of integer; m,n,i,j,lin,col:integer; function vizitat(k,lin, col:integer):boolean; begin vizitat:=false; for i:=1 to k-1 do if (sol[i,1]=lin) and (sol[i,2]=col) then vizitat:=true; end; procedure tipar(k,lin,col:integer); begin writeln('Solutie'); for i:=1 to k-1 do writeln(sol[i,1],',', sol[i,2]); if lin=0 then writeln('iesire prin nord') else if lin=m+1 then writeln('iesire prin sud') else if col=0 then writeln('iesire prin vest') else writeln('iesire prin est'); readln; end;</pre>	<pre>#include <iostream.h> int sol[100][2],l[10][10], m,n,i,j,lin,col; int vizitat(int k,int lin, int col) { int v=0; for (i=1;i<=k;i++) if (sol[i][0]==lin && sol[i][1]==col) v=1; return v; } void tipar(int k,int lin, int col) { cout << " Solutie " << endl; for (i=1;i<=k-1;i++) cout << sol[i][0] << " " << sol[i][1] << endl; if (lin==0) cout << "iesire prin nord" << endl; else if (lin==m+1) cout << "iesire prin sud" << endl; else if (col==0) cout << "iesire prin vest" << endl; else cout << "iesire prin est" << endl; }</pre>

```
procedure
back(k,lin,col:integer);
var i:integer;
begin
  if (lin=0) or (lin=m+1) or
    (col=0) or (col=n+1)
  then tipar(k,lin,col)
  else
    begin
      sol[k,1]:=lin;
      sol[k,2]:=col;
      for i:=1 to 4 do
        case i of
          1:if (l[lin,col]
            and 8<>0) and not
            vizitat(k,lin-1,col)
            then
              back(k+1,lin-1,col);
          2:if (l[lin,col]
            and 4<>0) and not
            vizitat(k,lin,col+1)
            then
              back(k+1,lin,col+1);
          3:if (l[lin,col]
            and 2<>0) and not
            vizitat(k,lin+1,col)
            then
              back(k+1,lin+1,col);
          4:if (l[lin,col]
            and 1<>0) and not
            vizitat(k,lin,col-1)
            then
              back(k+1,lin,col-1)
            end; {case}
      end;
    begin
      write('M=');
      readln(m);
      write('N=');
      readln(n);
      for i:=1 to m do
        for j:=1 to n do
          begin
            write('l[',i,',',j,']=');
            readln(l[i,j]);
          end;
      write('lin=');
      readln(lin);
      write('col=');
      readln(col);
      back(1,lin,col);
    end;
  
```

```
void back(int k, int lin,
          int col)
{ if (lin==0 || lin==m+1 ||
    col==0 || col==n+1)
  tipar(k,lin,col);
else
{
  sol[k][0]=lin;
  sol[k][1]=col;
  for (int i=1;i<=4;i++)
    switch(i)
    {
      case 1:
        if (l[lin][col] & 8 &&
            ! vizitat(k, lin-1,col))
          back(k+1,lin-1,col);
        break;
      case 2:
        if (l[lin][col] & 4 &&
            ! vizitat(k, lin,col+1))
          back(k+1,lin,col+1);
        break;
      case 3:
        if (l[lin][col] & 2 &&
            ! vizitat(k, lin+1,col))
          back(k+1,lin+1,col);
        break;
      case 4:
        if (l[lin][col] & 1 &&
            ! vizitat(k, lin,col-1))
          back(k+1,lin,col-1);
        break;
    }
}
main()
{ cout<<"M=";
  cin>>m;
  cout<<"N=";
  cin>>n;
  for (i=1;i<=m;i++)
    for(j=1;j<=n;j++)
      { cout<<l["<<i<<",
                  <<j<<"]="";
        cin>>l[i][j];
      }
  cout<<"Linie=";
  cin>>lin;
  cout<<"Coloana= ";
  cin>>col;
  back(1,lin,col);
}
```

? **Intrebare.** Cum s-ar putea găsi un drum de lungime minimă de la camera inițială către ieșirea din labirint? Prima idee care ne vine în minte este să generăm toate ieșirile, ca în program, pentru a o putea depista pe cea de lungime minimă. El bine, răspunsul nu este satisfăcător. Nu uitați, tehnica backtracking necesită un timp exponențial. Problema se poate rezolva cu mult mai eficient. Dar, pentru asta, trebuie să studiem teoria grafurilor. Toate la timpul lor...



Exerciții

- Adaptați rezolvarea pentru un labirint în care fiecare căsuță reține valoarea 1 sau 0 (1 semnificând căsuță plină, prin care nu se poate trece, iar 0 căsuță liberă, pe unde se poate trece). Ca și în problema prezentată, deplasarea se poate face dintr-o căsuță în orice altă căsuță alăturată, orizontal sau vertical, cu condiția ca ea să existe și să fie liberă. Validați poziția inițială a omului (lin, col), astfel încât aceasta să corespundă unei căsuțe libere. Estimați spațiul de memorie utilizat în această variantă.
- Realizați o variantă a rezolvării de la 1, adăugând câte o linie sau coloană suplimentară pe fiecare margine a labirintului, astfel încât să nu se mai testeze ca celula în care se trece să existe. Plasarea într-o celulă de pe margine este echivalentă cu obținerea unei soluții.

4.5.5. Problema bilei

□ Enunț. Se dă un teren sub formă de matrice cu m linii și n coloane. Fiecare element al matricei reprezintă un subteren cu o anumită altitudine dată de valoarea reținută de element (număr natural). Într-un astfel de subteren, de coordonate (lin, col) se găsește o bilă. Știind că bila se poate deplasa în orice porțiune de teren aflată la nord, est, sud sau vest, de altitudine strict inferioară porțiunii pe care se găsește bila. Se cere să se găsească toate posibilitățile ca bila să părăsească terenul.

Exemplu. Fie terenul alăturat. Inițial, bila se află în subterenul de coordonate $(2, 2)$. O posibilitate de ieșire din teren este dată de drumul: $(2, 2)$, $(2, 3)$, $(3, 3)$, $(3, 4)$. În program, altitudinile subterenului vor fi reținute de matricea t .

6	8	9	3
9	7	6	3
5	8	5	4
8	3	7	1

✓ Analiza problemei și rezolvarea ei. Problema seamănă cu cea anterioară, deci putem gândi că dacă înlocuim testul de intrare într-o cameră cu cel de altitudine mai mică, am rezolvat-o! Este adevarat, se obține o rezolvare, dar se poate și mai ușor. Să analizăm: mai este necesar să testăm dacă bila nu a ajuns pe un teren pe unde a mai trecut? Nu, deoarece, la fiecare pas, bila se deplasează pe un teren de altitudine strict inferioară. Prin urmare, problema este mai ușoară decât precedenta.

În rest, vom propune o rezolvare în care sol este o matrice cu 3 coloane și un număr mare de linii. Astfel, $sol(k, 1)$ va reține direcția în care pleacă bila (1 pt nord, 2 pentru est, 3 pentru sud și 4 pentru vest), $sol(k, 2)$ va reține linia subterenului, iar $sol(k, 3)$ va reține coloana subterenului.



Exercițiu. Arătați, prin reprezentare grafică, modul de funcționare a algoritmului, pe baza structurii de date propusă.

Varianta Pascal	Varianta C++
<pre> var sol:array [1..100,1..3] of integer; t:array [0..10,0..10] of integer; m,n,i,j,lin,col:integer; procedure tipar(k:integer); begin writeln('Solutie'); for i:=1 to k-1 do writeln(sol[i,2],' ',sol[i,3]); end; procedure back(k,lin,col:integer); begin if (lin=0) or (lin=m+1) or (col=0) or (col=n+1) then tipar(k) else begin sol[k,1]:=0; sol[k,2]:=lin; sol[k,3]:=col; while sol[k,1]<4 do begin sol[k,1]:=sol[k,1]+1; case sol[k,1] of 1:if t[lin-1,col]< t[lin,col] then back(k+1,lin-1,col); 2:if t[lin,col+1]< t[lin,col] then back(k+1,lin,col+1); 3:if t[lin+1,col]< t[lin,col] then back(k+1,lin+1,col+1); 4:if t[lin,col-1]< t[lin,col] then back(k+1,lin,col-1); end; {case} end end; begin write('M='); readln(m); write('N='); readln(n); </pre>	<pre> #include <iostream.h> int sol[100][3],t[10][10], m,n,i,j,lin,col; void tipar(int k) { cout<<"Solutie "<<endl; for(i=1;i<=k-1;i++) cout<<sol[i][1]<<" " <<sol[i][2]<<endl; } void back(int k, int lin, int col) { if (lin==0 lin==m+1 col==0 col==n+1) tipar(k); else { sol[k][0]=0; sol[k][1]=lin; sol[k][2]=col; while (sol[k][0]<4) { sol[k][0]++; switch(sol[k][0]) { case 1: if(t[lin-1][col]<t[lin][col]) back(k+1,lin-1,col); break; case 2: if(t[lin][col+1]<t[lin][col]) back(k+1,lin,col+1); break; case 3: if(t[lin+1][col]<t[lin][col]) back(k+1,lin+1,col); break; case 4: if(t[lin][col-1]<t[lin][col]) back(k+1,lin,col-1); break; } } } main() { cout<<"M="; cin>>m; cout<<"N="; cin>>n; } </pre>

```

for i:=1 to m do
  for j:=1 to n do
    begin
      write('t[',i,',',',j,
            ']=');
      readln(t[i,j])
    end;
  write('lin=');readln(lin);
  write('col=');readln(col);
  back(1,lin,col);
end.

```

```

for (i=1;i<=m;i++)
  for (j=1;j<=n;j++)
  { cout<<"t["<<i<<","<<j<<"]=", 
    cin>>t[i][j];
  }
cout<<"lin="; cin>>lin;
cout<<"col="; cin>>col;
back(1,lin,col);
}

```



Exercițiu. Modificați programul astfel încât să se afișeze și direcția în care bila părăsește terenul.

4.5.6. Săritura calului

Enunț. Se consideră o tablă de șah $n \times n$ și un cal plasat în colțul din stânga, sus. Se cere să se afișeze o posibilitate de mutare a acestei piese de șah, astfel încât să treacă o singură dată prin fiecare pătrat al tablei. Alăturat, observați o soluție pentru $n=5$.

1	16	11	20	3
10	21	2	17	12
15	24	19	4	7
22	9	6	13	18
25	14	23	8	5

Analiza problemei și rezolvarea ei. Fiind dată o poziție în care se găsește calul, acesta poate sări în alte 8 poziții. Pentru a scrie mai puțin cod, cele 8 poziții sunt reținute de vectorii constanți x și y . Astfel, dacă lin și col sunt coordonatele poziției unde se găsește calul, acesta poate fi mutat în:

$(lin+x[0], col+y[0]) \dots (lin+x[7], col+y[7])$.

Rețineți acest procedeu! De altfel, acesta poate fi folosit pentru problemele deja prezentate. Matricea t reține pozițiile pe unde a trecut calul. În rest, s-a procedat ca la problema anterioară.

Varianta Pascal	Varianta C++
<pre> const x:array[1..8] of integer=(-1,1,2,2,1,-1,-2,-2); y:array[1..8] of integer=(2,2,1,-1,-2,-2,-1,1); var n:integer; st: array[1..1000,1..2] of integer; t: array[-1..25,-1..25] of integer; procedure back(k,lin, col:integer); var i,linie,coloana:integer; </pre>	<pre> #include <iostream.h> #include <stdlib.h> const int x[8]={-1,1,2,2,1,-1,-2,-2}; const int y[8]={2,2,1,-1,-2,-2,-1,1}; int n,sol[1000][2],t[25][25]; void back(int k,int lin, int col) { int linie,coloana,i; </pre>

```

begin
  if k=n*n then
  begin
    for i:=1 to k-1 do
      writeln(st[i,1],' ',
             st[i,2]);
    writeln(lin,' ',col);
    halt;
  end
  else
  begin
    st[k,1]:=lin; st[k,2]:=col;
    for i:=1 to 8 do
    begin
      linie:=lin+x[i];
      coloana:=col+y[i];
      if (linie<=n) and
         (linie>=1) and
         (coloana<=n) and
         (coloana>=1) and
         (t[linie,coloana]=0)
      then begin
        t[linie,coloana]:=1;
        back(k+1,linie,
              coloana);
        t[linie,coloana]:=0;
      end;
    end;
  end;
end;
begin
  write ('n='); readln(n);
  back(1,1,1);
end.

```

```

if (k==n*n)
{ for (i=1;i<=k-1;i++)
  cout<<sol[i][0]<< " "
    <<sol[i][1]<<endl;
  cout<<lin<< " "<<col;
  exit(EXIT_SUCCESS);
}
else
{ sol[k][0]=lin;
  sol[k][1]=col;
  for (i=0;i<=7;i++)
  { linie=lin+x[i];
    coloana=col+y[i];
    if (linie<=n && linie>=1
        && coloana<=n &&
        coloana>=1 &&
        t[linie][coloana]==0)
    {
      t[linie][coloana]=1;
      back(k+1,linie,
            coloana);
      t[linie][coloana]=0;
    }
  }
}
main()
{ cout<<"n=";
  cin>>n;
  back(1,1,1);
}

```

Probleme propuse

- Avem la dispoziție 6 culori: alb, galben, roșu, verde, albastru și negru. Să se precizeze toate drapelele tricolore care se pot proiecta, știind că trebuie respectate regulile:
 - orice drapel are culoarea din mijloc galben sau verde;
 - cele trei culori de pe drapel sunt distințe.
- Dintr-un grup de n persoane, dintre care p femei, trebuie formată o delegație de k persoane, din care 1 femei. Să se precizeze toate delegațiile care se pot forma.
- La o masă rotundă se așeză n persoane. Fiecare persoană reprezintă o firmă. Se dau k perechi de persoane care aparțin unor firme concurente. Se cere să se determine toate modalitățile de așezare la masă a persoanelor, astfel încât să nu stea alături două persoane de la firme concurente.

4. Se dă o permutare a primelor n numere naturale. Se cer toate permutările care se pot obține din aceasta astfel încât nici o succesiune de două numere, existentă în permutarea inițială, să nu mai existe în noile permutări.
5. Se cer toate soluțiile de așezare în linie a m câini și n pisici astfel încât să nu existe o pisică așezată între doi câini.
6. **Anagrame.** Se citește un cuvânt cu n litere. Se cere să se tipărească toate anagramele cuvântului citit. Se poate folosi algoritmul pentru generarea permutărilor?
7. Se dă primele n numere naturale. Dispunem de un algoritm de generare a combinațiilor de n elemente luate câte p pentru ele. Se consideră un vector cu n componente șiruri de caractere, unde, fiecare șir reprezintă numele unei persoane. Cum adaptați algoritmul de care dispuneți pentru a obține combinațiile de n persoane luate câte p ?
8. Se citeșc n numere naturale distincte. Se cere o submulțime cu p elemente astfel încât suma elementelor sale să fie maximă în raport cu toate submulțimile cu același număr de elemente.
9. Să se determine 5 numere de căte n cifre, fiecare cîfră putând fi 1 sau 2, astfel încât oricare dintre aceste 5 numere să coincidă exact în m poziții și să nu existe o poziție în care să apară aceeași cîfră în toate cele 5 numere.
10. Fiind dat un număr natural pozitiv n , se cere să se producă la ieșire toate descompunerile sale ca sumă de numere prime.
11. **"Attila și regele".** Un cal și un rege se află pe o tablă de șah. Unele câmpuri sunt "arse", pozițiile lor fiind cunoscute. Calul nu poate călca pe câmpuri "arse", iar orice mișcare a calului face ca respectivul câmp să devină "ars". Să se afle dacă există o succesiune de mutări permise (cu restricțiile de mai sus), prin care calul să poată ajunge la rege și să revină la poziția inițială. Poziția inițială a calului, precum și poziția regelui sunt considerate "nearse".
12. Se dă n puncte în plan prin coordonatele lor. Se cer toate soluțiile de unire a acestor puncte prin exact p drepte, astfel încât mulțimea punctelor de intersecție ale acestor drepte să fie inclusă în mulțimea celor n puncte.
13. Găsiți toate soluțiile naturale ale ecuației $3x+y+4xz=100$.
14. Să se ordeneze în toate modurile posibile elementele mulțimii $\{1, 2, \dots, n\}$ astfel încât numerele $i, i+1, \dots, i+k$ să fie unul după celălalt și în această ordine ($1=1, i+k \leq n$).
15. Se consideră o mulțime de n elemente și un număr natural k nenul. Să se calculeze căte submulțimi cu k elemente satisfac, pe rînd, condițiile de mai jos și să se afișeze aceste submulțimi.
 - conțin p obiecte date;
 - nu conțin nici unul din q obiecte date;
 - conțin exact un obiect dat, dar nu conțin un altul;
 - conțin exact un obiect din p obiecte date;

- conțin cel puțin un obiect din p obiecte date;
 - conțin r obiecte din p obiecte date, dar nu conțin alte q obiecte date.
16. Se dă un număr natural par n . Să se determine toate șirurile de n paranteze care se închid corect.

Exemplu: pentru $n=6$: ((()), (()), () () , () () , () ().
 17. Se dau n puncte albe și n puncte negre în plan, de coordonate întregi. Fiecare punct alb se unește cu căte un punct negru, astfel încât din fiecare punct, fie el alb sau negru, pleacă exact un segment. Să se determine o astfel de configurație de segmente astfel încât oricare două segmente să nu se intersecteze. Se citeșc n perechi de coordonate corespunzînd punctelor albe și n perechi de coordonate corespunzînd punctelor negre.
 18. Să se genereze toate permutările de n cu proprietatea că oricare ar fi $2 \leq i \leq n$, există $1 \leq j \leq i$ astfel încât $|v(i) - v(j)| = 1$. **Exemplu:** pentru $n=4$, permutările cu proprietatea de mai sus sunt:

2134, 2314, 3214, 2341, 3241, 3421, 4321.
 19. O trupă cu n actori își propune să joace o piesă cu A acte astfel încât:
 - oricare două acte să aibă distribuție diferită;
 - în orice act există, evident, cel puțin un actor;
 - de la un act la altul, vine un actor pe scenă sau pleacă un actor de pe scenă (distribuția a două acte consecutive diferă prin exact un actor).

Să se furnizeze o soluție, dacă există vreuna.
 20. Fiind dat un număr natural n și un vector v cu n componente întregi, se cere:
 - să se determine toate subșirurile crescătoare de lungime $[n/5]$;
 - să se calculeze $p(1) + p(2) + \dots + p(k)$, unde $p(k)$ reprezintă numărul subșirurilor crescătoare de lungime k .
 21. Pe malul unei ape se găsesc c canibali și m misionari. Ei urmează să treacă apa și au la dispoziție o barcă cu 2 locuri. Se știe că, dacă atât pe un mal, cât și pe celălalt avem mai mulți canibali decât misionari, misionarii sunt mâncăți de canibali. Se cere să se scrie un program care să furnizeze toate soluțiile de trecere a apei, astfel încât să nu fie mâncat nici un misionar.
 22. Se dă un careu sub formă de matrice cu m linii și n coloane. Elementele careului sunt litere. Se dă, de asemenea, un cuvânt. Se cere să se găsească în careu prefixul de lungime maximă al cuvântului respectiv. Regula de căutare este următoarea:
 - a) se caută litera de început a cuvântului;
 - b) litera următoare se caută printre cele 4 elemente învecinate cu elementul care conține litera de început, apoi printre cele 4 elemente învecinate cu elementul care conține noua literă, și.a.m.d.

23. Se dă coordonatele a n puncte din plan. Se cere să se precizeze 3 puncte care determină un triunghi de arie maximă. Ce algoritm vom folosi?
- Generarea aranjamentelor;
 - Generarea combinărilor;
 - Generarea permutărilor;
 - Generarea tuturor submulțimilor.
24. Fiind date numele a n soldați, ce algoritm vom utiliza pentru a lista toate grupele de căte k soldați? Se știe că într-o grupă, ordinea prezintă importanță.
- Generarea aranjamentelor;
 - Generarea combinărilor;
 - Generarea permutărilor;
 - Generarea tuturor partițiilor.
25. Fiind date n numere naturale, ce algoritm vom utiliza pentru a determina *eficient* o submulțime maximală de numere naturale distințe?
- se generează toate submulțimile și se determină o submulțime maximală care îndeplinește condiția cerută;
 - se generează toate partițiile și se caută o submulțime maximală care aparține unei partiții oarecare și care îndeplinește condiția cerută;
 - se compară primul număr cu al doilea, al treilea, al n -lea, al doilea cu al treilea, al patrulea, al n -lea, ... și atunci când se găsește egalitate se elimină un număr dintre ele.
 - nici un algoritm dintre cei de mai sus.
26. Dispunem de un algoritm care generează permutările prin backtracking. Primele două permutări afișate sunt: 321, 312. Care este următoarea permutare care va fi afișată?
- 321;
 - 123;
 - 213;
 - 231.

Indicații

6. Deși algoritmul este asemănător, nu este același, trebuie pusă o condiție suplimentară. De exemplu, în cuvântul "mama" nu se poate inversa a de pe poziția 2 cu a de pe poziția 4.

7. Dacă vectorul care reține numele persoanelor este v , în loc să se afișeze i , se va afișa $v[i]$.

23. b). 24. a). 25. d).

Explicație: primele două variante prezintă soluții exponentiale, a treia este în $O(n^2)$. Dar dacă sortăm numerele, atunci le putem afișa pe cele distințe dintr-o singură parcurgere. Sortarea se poate efectua în $O(n \times \log(n))$, iar parcurgerea în $O(n)$. Prin urmare, complexitatea este $O(n \times \log(n))$.

26. d).

Explicație: pe fiecare nivel al stivei se caută succesorii în ordinea $n, n-1, \dots, 1$.

Capitolul 5

Metoda Greedy

5.1. Generalități

Enunț general:

"Se consideră o mulțime A . Se cere o submulțime a sa astfel încât să fie îndeplinite anumite condiții (acestea diferă de la o problemă la alta)".

Uneori, pentru astfel de probleme, se poate folosi metoda **Greedy**.

Structura generală a unei aplicații **Greedy** este dată mai jos:

```
Sol:=∅;
Repetă
| Alege  $x \in A$ ;
| Dacă este Posibil atunci Sol←Sol+ $x$ ;
| Până când am obținut soluția
Afișează Sol;
```

Soluția este inițial vidă și, pe rând, se alege dintre elementele mulțimii A , element cu element, până când se obține soluția cerută.

- ⇒ De regulă, metoda **Greedy** rezolvă problema în $O(n^k)$.
- ⇒ Datorită faptului că soluția este construită pas cu pas, fără un mecanism de revenire ca la **Backtracking**, metoda se numește **Greedy** (în română, traducerea este "iacom").
- ⇒ Deși s-a prezentat forma generală a metodei, ea nu poate fi standardizată (să avem o unică rutină pe care s-o aplicăm la rezolvarea oricărei probleme).
- ⇒ Există destul de puține probleme pentru care se poate aplica această metodă.
- ⇒ Metoda **Greedy** se aplică în două cazuri:
 - 1) atunci când știm sigur că se ajunge la soluția dorită (avem la bază o demonstrație);
 - 2) atunci când nu dispunem de o soluție în timp polinomial, dar prin **Greedy** obținem o soluție acceptabilă, nu neapărat optimă.

5.2. Probleme pentru care metoda Greedy conduce la soluția optimă

5.2.1. Suma maximă

Enunț. Se consideră o mulțime de n numere reale. Se cere o submulțime a sa, cu un număr maxim de elemente, astfel încât suma elementelor sale să fie maximă.

Rezolvare. Nu este prea greu de observat că trebuie selectate doar elementele mai mari sau egale cu 0. Datorită faptului că se dorește ca submulțimea să aibă un număr maxim de elemente, se selecteză și elementele egale cu 0. Soluția este dată de vectorul B . Am căutat între elementele vectorului A și le-am ales numai pe cele mai mari sau egale cu 0.

Programul este următorul:

Varianța Pascal	Varianța C++
<pre>var A,B:array[1..100]of real; n,m,i:integer; procedure Greedy; begin for i:=1 to n do if A[i]>=0 then begin m:=m+1; B[m]:=A[i]; end; end; begin write ('n='); readln(n); for i:=1 to n do begin write('A[,i,]='); readln (A[i]); end; Greedy; for i:=1 to m do writeln(B[i]:4:2); end.</pre>	<pre>#include <iostream.h> float A[100],B[100]; int n,m,i; void Greedy() { for(i=1;i<=n;i++) if (A[i]>=0) { m++; B[m]=A[i]; } } main() { cout<<"n="; cin>>n; for(i=1;i<=n;i++) { cout<<"A["<<i<<"]="; cin>>A[i]; } Greedy(); for(i=1;i<=m;i++) cout<<B[i]<<" ";</pre>

5.2.2. Problema planificării spectacolelor

Enunț. Într-o sală, într-o zi, trebuie planificate n spectacole. Pentru fiecare spectacol se cunoaște intervalul în care se desfășoară: $[st, sf]$. Se cere să se planifice un număr maxim de spectacole, astfel încât să nu se suprapună.

Rezolvare. Vom construi o soluție după următorul algoritm:

P_1 Sortăm spectacolele după ora terminării lor.

P_2 Primul spectacol programat este cel care se termină cel mai devreme.

P_3 Alegem primul spectacol dintre cele care urmează în sir ultimului spectacol programat care îndeplinește condiția că începe după ce s-a terminat ultimul spectacol programat.

P_4 Dacă tentativa de mai sus a eșuat (nu am găsit un astfel de spectacol), algoritmul se termină; altfel, se programează spectacolul găsit și algoritmul se reia de la P_3 .

↓ Demonstrație

Fie $I_1 \ I_2 \ \dots \ I_k$ sirul spectacolelor alese de algoritm și $O_1 \ O_2 \ \dots \ O_s$ o soluție optimă.

Dacă $k > s$, înseamnă că $O_1 \ O_2 \ \dots \ O_s$ nu este soluție optimă.

Dacă $k = s$, înseamnă că algoritmul a obținut o soluție optimă.

Dacă $k < s$, atunci procedăm astfel:

În soluția optimă se înlocuiește O_1 cu I_1 . Înlocuirea este posibilă, întrucât I_1 este spectacolul care se termină cel mai devreme, deci sirul $O_2 \ O_3 \ \dots \ O_s$ poate să-i urmeze. În acest fel nu se afectează optimalitatea soluției. Soluția este: $I_1 \ O_2 \ \dots \ O_s$. Dar I_2 este spectacolul care începe după ce se termină I_1 și se termină cel mai devreme dintre cele care îl urmează lui I_1 . Aceasta înseamnă că în nici un caz nu se termină mai târziu decât O_2 . Din acest motiv, îl înlocuim în soluția optimă pe O_2 cu I_2 .

Prinț-un raționament asemănător, ajungem la soluția optimă $I_1 \ I_2 \ \dots \ I_k \ O_{k+1} \ \dots \ O_s$. Aceasta înseamnă că după I_k s-au mai putut selecta și alte spectacole; absurd, dacă ținem cont că, după I_k , nu se mai pot selecta alte spectacole.

Deci, $k = s$, q.e.d.

Programul este prezentat mai jos:

Varianta Pascal	Varianta C++
<pre> var s: array[1..2,1..10] of integer; o:array[1..10] of integer; n,i,h1,m1,h2,m2:integer; procedure sortare; var gata:boolean; m,i:integer; begin repeat gata:=true; for i:=1 to n-1 do if s[2,o[i]]>s[2,o[i+1]] then begin m:=o[i]; o[i]:=o[i+1]; o[i+1]:=m; gata:=false end until gata; end; begin write('n='), readln(n); for i:=1 to n do begin o[i]:=i; write('ora de inceput pentru spectacolul ',i, ' (hh mm)='); readln(h1,m1); s[1,i]:=h1*60+m1; write('ora de sfirsit pentru spectacolul ',i, ' (hh mm)='); readln(h2,m2); s[2,i]:=h2*60+m2; end; sortare; { greedy } writeln('ordinea spectacolelor este'); writeln(o[1]); for i:=2 to n do if s[1,o[i]]>=s[2,o[i-1]] then writeln (o[i]); end. </pre>	<pre> #include <iostream.h> int s[2][10],o[10],n,i, h1,m1,h2,m2,ora; void sortare() { int gata,m,i; do { gata=1; for (i=1;i<=n-1;i++) if (s[1][o[i]]> s[1][o[i+1]]) { m=o[i]; o[i]=o[i+1]; o[i+1]=m; gata=0; } while (!gata); } main() { cout<<"n="; cin>>n; for (i=1;i<=n;i++) { o[i]=i; cout<<"ora de inceput pentru spectacolul "<<i <<" (hh mm)="; cin>>h1>>m1; s[0][i]=h1*60+m1; cout<<"ora de sfirsit pentru spectacolul "<<i <<" (hh mm)="; cin>>h2>>m2; s[1][i]=h2*60+m2; } sortare(); cout<<"ordinea spectacolelor este "<<endl<<o[1]<<endl; ora=s[1][o[1]]; for (i=2;i<=n;i++) { if (s[0][o[i]]>=ora) { cout<<o[i]<<endl; ora=s[1][o[i]]; } } } } </pre>

5.2.3. Problema rucsacului (cazul continuu)

□ **Enunț.** O persoană are un rucsac cu ajutorul căruia poate transporta o greutate maximă **G**. Persoana are la dispoziție **n** obiecte și cunoaște pentru fiecare obiect greutatea și câștigul care se obține în urma transportului său la destinație.

Se cere să se precizeze ce obiecte trebuie să transporte persoana în aşa fel încât câștigul să fie maxim.

O precizare în plus transformă această problemă în alte două probleme distincte. Această precizare se referă la faptul că obiectele pot fi sau nu tăiate pentru transportul la destinație. În prima situație, problema poartă numele de **problema continuă a rucsacului**, iar în a doua avem **problema discretă a rucsacului**. Aceste două probleme se rezolvă diferit, motiv pentru care sunt prezentate separat.

Varianta continuă a problemei rucsacului este tratată în acest paragraf, iar cea discretă va fi tratată cu ajutorul programării dinamice.

✓ **Rezolvare.** Algoritmul este următorul:

- se calculează, pentru fiecare obiect în parte, eficiența de transport rezultată prin împărțirea câștigului la greutate (de fapt, acesta reprezintă câștigul obținut prin transportul unității de greutate);
- obiectele se sortează în ordine descrescătoare a eficienței de transport și se preiau în calcul în această ordine;
- câștigul inițial va fi 0, iar greutatea rămasă de încărcat va fi **G**;
- atât timp cât nu a fost completată greutatea maximă a rucsacului și nu au fost luate în considerare toate obiectele, se procedează astfel:
 - dintre obiectele neîncărcate se selectează acela cu cea mai ridicată eficiență de transport și avem două posibilități:
 - obiectul începe în totalitate în rucsac, deci se scade, din greutatea rămasă de încărcat, greutatea obiectului, iar la câștig se cumulează câștigul datorat transportului acestui obiect; se tipărește 1, în sensul că întregul obiect a fost încărcat;
 - obiectul nu începe în totalitate în rucsac, caz în care se calculează ce parte din el poate fi transportată, se cumulează câștigul obținut cu transportul acestei părți din obiect, se tipărește procentul care s-a încărcat din obiect, iar greutatea rămasă de încărcat devine 0.



Dăm un exemplu numeric. Greutatea care poate fi transportată cu ajutorul rucsacului este 3. Avem la dispoziție 3 obiecte. Greutatea și câștigul pentru fiecare obiect sunt prezentate mai jos:

C	2	4	6
G	2	1	3

Eficiența de transport este 1 pentru primul obiect, 4 pentru al doilea și 2 pentru al treilea. În concluzie, obiectul 2 se încarcă în întregime în rucsac, obținând un câștig de 4 și rămâne o capacitate de 2 unități de greutate. Se încarcă $\frac{2}{3}$ din obiectul 3 (care este al doilea în ordinea eficienței de transport) pentru care se obține câștigul 4. Câștigul obținut în total este 8.

Se remarcă strategia GREEDY prin alegerea obiectului care va fi transportat, alegere asupra căreia nu se revine.

Varianța Pascal	Varianța C++
<pre> type vector=array [1..9] of real; var c,g,ef:vector; n,i,man1:integer; gv,man,castig:real; inv:boolean; ordine:array [1..9] of integer; begin write('Greutatea ce poate fi transportata='); readln(gv); write('Numar de obiecte='); readln(n); for i:=1 to n do begin write('c['',i,'']='); readln(c[i]); write('g['',i,'']='); readln(g[i]); ordine[i]:=i; ef[i]:=c[i]/g[i]; end; repeat inv:=false; for i:=1 to n-1 do if ef[i]<ef[i+1] then begin man:=ef[i]; ef[i]:=ef[i+1]; ef[i+1]:=man; man:=c[i]; c[i]:=c[i+1]; c[i+1]:=man; man:=g[i]; end; until inv; end. </pre>	<pre> #include <iostream.h> double c[9],g[9],ef[9],gv,man, castig; int n,i,man1,inv,ordine[9]; main() { cout<<"Greutatea ce poate fi transportata=", cin>>gv; cout<<"Numar de obiecte="; cin>>n; for (i=1;i<=n;i++) { cout<<"c["<<i<<"]="; cin>>c[i]; cout<<"g["<<i<<"]="; cin>>g[i]; ordine[i]=i; ef[i]=c[i]/g[i]; } do { inv=0; for (i=1;i<=n-1;i++) if (ef[i]<ef[i+1]) { man=ef[i]; ef[i]=ef[i+1]; ef[i+1]=man; man=c[i]; c[i]=c[i+1]; c[i+1]=man; man=g[i]; g[i]=g[i+1]; g[i+1]=man; inv=1; } man1=ordine[i]; ordine[i]=ordine[i+1]; ordine[i+1]=man1; } while (inv); } </pre>

```

i=1;
while (gv>0 && i<=n)
{
  if (gv>g[i])
    {
      cout<<"Obiectul "
          <<ordine[i]<<'
          <<i<<endl;
      gv-=g[i];
      castig+=c[i];
    }
  else
    {
      cout<<"Obiectul "
          <<ordine[i]<<'
          <<gv/g[i]<<endl;
      castig+=c[i]*gv/g[i];
      gv=0;
    }
  i++;
}
cout<<"Castig total="<<castig;
}

```

5.2.4. O problemă de maxim

Enunț. Se dau o mulțime **A** cu m numere întregi nenule și o mulțime **B** cu $n \geq m$ numere întregi nenule. Se cere să se selecteze un sir cu m elemente din **B**, x_1, x_2, \dots, x_m astfel încât expresia următoare să fie maximă:

$$E=a_1x_1+a_2x_2+\dots+a_nx_n,$$

unde, a_1, a_2, \dots, a_n sunt elementele mulțimii **A** într-o anumită ordine pe care trebuie să-o determinați.

Rezolvare. Vom sorta crescător elementele mulțimii **A** și obținem a_1, a_2, \dots, a_n și pe cele ale mulțimii **B**. **E** maxim va fi:

$$E_{\max}=a_1b_{n-m+1}+a_2b_{n-m+2}+\dots+a_mb_n.$$

Cazul 1: $m=n$. Se presupune A sortat crescător. Dacă B nu este sortat crescător, fie prima inversare pe care o face algoritmul de sortare prin inversare (metoda bulelor): b_i cu b_{i+1} ($b_i > b_{i+1}$).

Inițial, vectorul B este

$$B = (b_1, b_2, \dots, b_i, b_{i+1}, \dots, b_m) \text{ și } E_1 = a_1 b_1 + \dots + a_i b_i + a_{i+1} b_{i+1} + \dots + a_m b_m.$$

După inversare, B este:

$$B = (b_1, b_2, \dots, b_{i+1}, b_i, \dots, b_m) \text{ și } E_2 = a_1 b_1 + \dots + a_i b_{i+1} + a_{i+1} b_i + \dots + a_m b_m.$$

$$E_2 - E_1 = a_i b_{i+1} + a_{i+1} b_i - a_i b_i - a_{i+1} b_{i+1} = b_{i+1}(a_i - a_{i+1}) - b_i(a_i - a_{i+1}) = (b_{i+1} - b_i)(a_i - a_{i+1}) \geq 0.$$

Cu alte cuvinte, orice astfel de inversare, efectuată de sortarea prin inversare, mărește valoarea lui E . Prin urmare, valoarea maximă se obține atunci când vectorul este sortat crescător.

Cazul 2: $m > n$.

Dacă $B = (b_1, b_2, \dots, b_m)$ și $B' = (b'_1, b'_2, \dots, b'_m)$, $b'_1 \geq b_1$, $b'_2 \geq b_2, \dots, b'_m \geq b_m$

$$E' - E = a_1(b'_1 - b_1) + a_2(b'_2 - b_2) + \dots + a_m(b'_m - b_m) \geq 0.$$

Aceasta justifică alegerea ultimelor m elemente din vectorul B sortat.

Varianta Pascal	Varianta C++
<pre>type vector=array[1..10] of integer; var A,B:vector; m,n,i,E:integer; procedure Sort(k:integer; var X:vector); var Inversari:boolean; man:integer; begin repeat Inversari:=false; for i:=1 to k-1 do if X[i]>X[i+1] then begin man:=X[i]; X[i]:=X[i+1]; X[i+1]:=man; Inversari:=true; end; until not Inversari; end;</pre>	<pre>#include <iostream.h> int A[30],B[30],m,n,i,E; void Sort(int k, int X[20]) { int inversari,man; do { inversari=0; for(i=1;i<=k-1;i++) if (X[i]>X[i+1]) { man=X[i]; X[i]=X[i+1]; X[i+1]=man; inversari=1; } } while (inversari); }</pre>

```
begin
write('M=');readln(m);
for i:=1 to m do readln(A[i]);
write('N=');readln(N);
for i:=1 to n do readln(B[i]);
Sort(m,A);
Sort(n,B);
for i:=1 to m do
E:=E+A[i]*B[n-m+i];
writeln ('Emax=', E);
end.

main()
{ cout<<"M="; cin>>m;
for(i=1;i<=m;i++) cin>>A[i];
cout<<"N="; cin>>n;
for(i=1;i<=n;i++) cin>>B[i];
Sort(m,A);
Sort(n,B);
for(i=1;i<=m;i++)
E+=A[i]*B[n-m+i];
cout<<"Emax="<<E;
}
```

5.3. Greedy heuristic

Există probleme pentru care se cunosc numai algoritmi exponențiali de rezolvare a lor. Aceștia sunt, după cum știți, inacceptabili în practică. Totuși, problemele există și, de multe ori, avem nevoie de o rezolvare. În astfel de cazuri, una dintre soluții este aceea de a aplica metoda **Greedy**, care va obține o soluție, nu neapărat optimă, dar acceptabilă. Un algoritm care conduce la o soluție acceptabilă, dar nu optimă, se numește heuristică, iar dacă acesta este de tip **Greedy**, se numește **Greedy heuristic**. Exemplul care urmează vă vor lămuri.

5.3.1. Plata unei sume întotdeauna număr minim de bancnote

Enunț. Se dau n tipuri de bancnote de valori b_1, b_2, \dots, b_m (numere naturale strict mai mari ca 0). Din fiecare tip se dispune de un număr nelimitat de bancnote. De asemenea, se știe că vom avea întotdeauna bancnota cu valoarea 1. Fiind dată o sumă s , număr natural, se cere ca aceasta să fie plătită prin utilizarea unui număr minim de bancnote.

Rezolvare. Algoritmul este simplu: se sortează bancnotele în ordinea descrescătoare a valorii lor. Suma este plătită, cât este posibil, cu bancnota din primul tip, apoi cu bancnota de al doilea tip, și.m.d., până când este achitată în totalitate.

Exemplu: $s=67$. Avem 3 tipuri de bancnote: de 10, de 5 și de 1. Soluția va fi: 6 bancnote de 10, 1 bancnotă de 5 și 2 bancnote de 1.

Algoritmul nu întoarce întotdeauna soluția optimă. Pentru $s=10$ și 3 tipuri de bancnote de valori 4, 3, 1, algoritmul va da soluția: 2 bancnote de 4 și 2 bancnote de 1, în total 4 bancnote. În realitate, soluția optimă este dată de 1 bancnotă de 4 și 2 de 3, adică 3 bancnote...

Dacă am dori să evităm acest algoritm, care nu conduce întotdeauna la soluția optimă, putem aplica metoda **Backtracking**. Dar algoritmul este exponențial...

Varianța Pascal	Varianța C++
<pre> type vector=array[1..10] of integer; var B:vector; n,i,S:integer; procedure Sort(k:integer; var X:vector); var Inversari:boolean; man:integer; begin repeat Inversari:=false; for i:=1 to k-1 do if X[i]<X[i+1] then begin man:=X[i]; X[i]:=X[i+1]; X[i+1]:=man; Inversari:=true; end; until not inversari; end; begin write('S='); readln(S); write ('n='); readln(n); for i:=1 to n do begin write ('Bancnote de valoarea '); readln(B[i]); end; Sort(n,B); i:=1; while (S) { if (S/B[i]) { cout<<"S="; cout<<"N="; for(i=1;i<=n;i++) { cout<<"Bancnote de val. "; cin>>B[i]; } Sort(n,B); i=1; while (S) { if (S/B[i]) { cout<<S/B[i]<< " Bancnote de valoare "<<B[i]<<endl; S-=S/B[i]*B[i]; } i++; } } end; </pre>	<pre> #include <iostream.h> int B[30],n,i,S; void Sort(int k, int X[20]) { int inversari,man; do { inversari=0; for(i=1;i<=k-1;i++) if (X[i]<X[i+1]) { man=X[i]; X[i]=X[i+1]; X[i+1]=man; inversari=1; } } while (inversari); main() { cout<<"S="; cout<<"N="; for(i=1;i<=n;i++) { cout<<"Bancnote de val. "; cin>>B[i]; } Sort(n,B); i=1; while (S) { if (S/B[i]) { cout<<S/B[i]<< " Bancnote de valoare "<<B[i]<<endl; S-=S/B[i]*B[i]; } i++; } } </pre>

5.3.2. Săritura calului

Enunț. Se dă o tablă de săh cu dimensiunea $n \times n$. Un cal se găsește în linia 1 și coloana 1. Găsiți un sir de mutări ale calului astfel încât acesta să acopere întreaga tablă fără a trece printr-o căsuță de două ori.

Rezolvare. Problema este binecunoscută, am tratat-o la **Backtracking**. Din păcate, rezultatele erau nesatisfăcătoare. Pentru $n=6$, se afișa imediat soluția, pentru $n=7$ se aștepta ceva, iar pentru $n=8\dots$

Pentru această problemă vom prezenta un algoritm euristic extrem de eficient. În ce constă?

La fiecare pas, alegem acea mutare care așează calul în poziția cel mai greu accesibilă la pasul următor. Fie calul în poziția (1, c). Teoretic, din acea poziție, calul poate fi mutat în alte 8 poziții. Desigur, nu toate sunt accesibile, deoarece pentru unele dintre ele calul părăsește tabla, iar pentru altele se ajunge în poziții deja vizitate. Dintre toate pozițiile în care se poate ajunge, se alege cea care este cât mai izolată. Vezi funcția **Numar**.

Cu această euristică, rezultatele sunt spectaculoase. De exemplu, pentru $n=10$ soluția se afișează instantaneu.. Însă... nu avem o demonstrație pentru aceasta, deci oricând este posibil să găsim o anumită valoare a lui n pentru care algoritmul nu furnizează nici o soluție. Rulați programul următor și vă veți convinge!

Varianța Pascal	Varianța C++
<pre> const x:array[1..8] of integer=(-1,1,2,2, 1,-1,-2,-2); y:array[1..8] of integer=(2,2,1,-1,-2,-2,-1,1); var t: array[-1..50,-1..50] of integer; Mutari,i,j,n:integer; function Numar(l,c:integer) :integer; var nr,i:integer; begin nr:=0; for i=0;i<=7;i++ if (l+x[i]>=1 && l+x[i]<=n && c+y[i]>=1 && c+y[i]<=n && t[l+x[i]][c+y[i]]==0 then nr:=nr+1; Numar:=nr; end; </pre>	<pre> #include <iostream.h> const int x[8]={-1,1,2,2,1,-1,-2,-2}; const int y[8]={2,2,1,-1,-2,-2,-1,1}; int t[50][50],Mutari,i,j,n; int Numar(int l, int c) { int nr=0,i; for (i=0;i<=7;i++) if (l+x[i]>=1 && l+x[i]<=n && c+y[i]>=1 && c+y[i]<=n && t[l+x[i]][c+y[i]]==0 nr++; return nr; } </pre>

```

procedure Mut(l,c:integer);
var i,min,v,linie,
    coloana:integer;
    gasit:boolean;

begin
  t[1,c]:=Mutari+1;
  gasit:=false;
  min:=9;
  for i:=1 to 8 do
    if t[l+x[i],c+y[i]]=0
    then
      begin
        v:=Numar(l+x[i],c+y[i]);
        if v<min then
          begin
            min:=v;
            linie:=l+x[i];
            coloana:=c+y[i];
            gasit:=true;
          end;
      end;
  if gasit then
    begin
      Mutari:=Mutari+1;
      Mut(linie,coloana)
    end
  end;

begin
  write ('n='); readln(n);
  for i:=0 to n+1 do
  begin
    t[0,i]:=1;
    t[-1,i]:=1;
    t[n+1,i]:=1;
    t[n+2,i]:=1;
    t[i,0]:=1;
    t[i,-1]:=1;
    t[i,n+1]:=1;
    t[i,n+2]:=1;
  end;
  Mut(1,1);
  if Mutari=n*n-1
  then
    for i:=1 to n do
    begin
      for j:=1 to n do
        write(t[i,j],' ');
      writeln;
    end
  else
    writeln('Tentativa Esuata');
end.

```

```

void Mut(int l, int c)
{ int i,min,v,linie,
  coloana,gasit;
  t[1][c]=Mutari+1;
  gasit=0;
  min=9;
  for (i=0;i<=7;i++)
    if (l+x[i]>=1 &&
        l+x[i]<=n &&
        c+y[i]>=1 &&
        c+y[i]<=n &&
        t[l+x[i]][c+y[i]]==0)
    { v=Numar(l+x[i],c+y[i]);
      if (v<min)
        { min=v;
          linie=l+x[i];
          coloana=c+y[i];
          gasit=1;
        }
      if (gasit)
        { Mutari++;
          Mut(linie,coloana);
        }
    }
  main()
  { cout<<"n="; cin>>n;
    t[1][1]=1;
    if (Mutari==n*n-1)
      for(i=1;i<=n;i++)
        { for(j=1;j<=n;j++)
          cout<<t[i][j]<<" ";
          cout<<endl;
        }
    else cout<<"Tentativa esuata";
  }
}

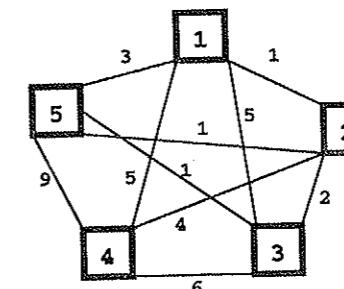
```

5.3.3. Problema comis–voiajorului

Enunț. Un comis–voiajor pleacă dintr-un oraș, trebuie să viziteze un număr de orașe și să se întoarcă în orașul de unde a plecat cu efort minim. Orice oraș i este legat printr-o șosea de orice alt oraș și printr-un drum de $A[i,j]$ kilometri. Se cere traseul pe care trebuie să-l urmeze comis–voiajorul, astfel încât să parcurgă un număr minim de kilometri.

! În practică, s-ar putea ca două orașe să nu fie unite printr-o șosea. Asta nu face inutilizabilă actuala problemă, pentru că se poate considera că orașele sunt unite printr-o șosea cu ∞ km (pentru calculator, un număr foarte mare).

Ex: Fie orașele de mai jos și matricea care reține distanțele dintre ele:



$$A = \begin{pmatrix} 0 & 1 & 5 & 5 & 3 \\ 1 & 0 & 2 & 4 & 1 \\ 5 & 2 & 0 & 6 & 1 \\ 5 & 4 & 6 & 0 & 9 \\ 3 & 1 & 1 & 9 & 0 \end{pmatrix}$$

Figura 5.1. Exemplu de hartă pentru problema comis–voiajorului

Rezolvare. Desigur, problema se poate rezolva prin **Backtracking**, generând toate drumurile și selectându-l pe cel de lungime minimă. Dar acest algoritm este exponențial. Atunci renunțăm la soluția optimă și ne mulțumim cu una "suficient de bună".

Ideea este extrem de simplă: se alege un oraș de pornire. La fiecare pas se selectează un alt oraș, prin care nu s-a mai trecut și aflat la distanță minimă față de orașul de pornire. Algoritmul se încheie atunci când am selectat toate orașele. Vectorul s va reține orașele deja selectate.

De exemplu, dacă se pornește cu orașul 1, se va obține un drum de lungime 14 care trece prin orașele: 1, 2, 5, 3, 4 și 1.

Varianta Pascal	Varianta C++
<pre> var S:array[1..9] of integer; A:array[1..9,1..9] of integer; n,i,j,v,vs,vs1,min,cost:integer; </pre>	<pre> #include <iostream.h> int S[10],A[10][10],n,i,j, v,vs,vs1,min,cost; </pre>

```

begin
  write('Numar noduri=');
  readln(n);
  {citesc matricea}
  for i:=1 to n do
    for j:=i+1 to n do
      begin
        write('A[,i,',',j,]=');
        readln(A[i,j]);
        A[j,i]:=A[i,j];
      end;
  {Pentru fiecare nod}
  write ('Nod de pornire ');
  readln(v);
  S[v]:=1;
  vs1:=v;
  write('Drumul trece prin ');
  for i:=1 to n-1 do
    begin
      min:=30000;
      for j:=1 to n do
        if (A[v,j]<>0) and (S[j]=0)
          and (min>a[v,j])
        then
          begin
            min:=A[v,j];
            vs:=j;
          end;
      cost:=cost+A[v,vs];
      write(vs,' ');
      S[vs]:=1;
      v:=vs;
    end;
  cost:=cost+A[v1,v];
  writeln('Cost=',Cost);
end.

```

```

main()
{
  cout<< "Numar noduri=";
  cin>>n;
  //citesc matricea
  for (i=1;i<=n;i++)
    for (j=i+1;j<=n;j++)
      {
        cout<<"A["<<i
              <<","<<j<<"]=";
        cin>>A[i][j];
        A[j][i]=A[i][j];
      }
  //Pentru fiecare nod
  cout<< "Nod de pornire ";
  cin>>v;
  S[v]=1;
  vs1=v;
  cout<< "Drumul trece prin ";
  for (i=1;i<=n-1;i++)
  {
    min=30000;
    for (j=1;j<=n;j++)
      if (A[v][j]!=0 && S[j]==0
          && min>A[v][j])
        {
          min=A[v][j];
          vs=j;
        }
    cost+=A[v][vs];
    cout<<vs<< " ";
    S[vs]=1;
    v=vs;
  }
  cost+=A[v1][v];
  cout<<endl<< "Cost=" <<cost;
}

```

Probleme propuse

1. În cazul unei mulțimi cu n numere reale, care este complexitatea algoritmului pentru selectarea unei mulțimi cu număr maxim de elemente pentru care suma elementelor este maximă?

- a) $O(n)$; b) $O(2n)$; c) $O(n^2)$; d) $O(2^n)$.

2. Care este complexitatea algoritmului prezentat în carte care rezolvă “**problema spectacolelor**”?

- a) $O(n)$; b) $O(\log(n))$; c) $O(n^2)$; d) $O(2^n)$.

3. Scrieți un program care rezolvă problema spectacolelor și are o complexitate mai mică decât programul prezentat.

4. Care este complexitatea algoritmului prezentat care rezolvă problema continuă a rucsacului și cum putem modifica acest algoritm pentru a avea o complexitate mai “bună”?

5. Rezolvați **problema rucsacului**, cazul continuu, în cazul în care la plecare, se dispune, din fiecare produs, de cantități nelimitate.

6. O întreprindere care fabrică produse alimentare (unt, marmeladă, ulei, etc) dispune de suma s pentru fabricarea anumitor produse. Se știe că există n produse care pot fi fabricate și pentru fiecare produs se cunoaște costul obținerii unui kg (litru) din produsul respectiv, precum și prețul de vânzare al unui kg (litru) din produsul respectiv. Scrieți un program care determină ce produse pot să fie fabricate de întreprindere și în ce cantitate, astfel încât să se obțină o sumă maximă la vânzarea lor.

7. Rezolvați prin **Backtracking** problema plății unei sume în număr minim de bancnote. Comparați soluțiile cu cele obținute prin **Greedy heuristică**, din punct de vedere al corectitudinii lor și din punct de vedere al timpului necesar pentru obținerea lor.

8. Care este complexitatea programului care rezolvă prin **Greedy heuristică** problema comis-voiajorului?

9. Scrieți un program care rezolvă prin **Greedy heuristică** problema comis-voiajorului, dar care caută cea mai bună soluție pornind de la fiecare vârf în parte. Care este complexitatea acestui algoritm?

10. **Codul Gray.** Pentru primele 8 numere naturale, în tabelul de mai jos, observați numărul scris în baza 10, în binar și în codul **Gray**. După cum puteți observa, codul **Gray** se caracterizează prin faptul că reprezentarea a două numere consecutive diferă exact cu o poziție binară.

Număr	În cod binar	În cod Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Fiind dat un număr natural în binar, $b_1b_2\dots b_n$, scrieți un subprogram care convertește numărul în cod Gray, $g_1g_2\dots g_n$, după următorul algoritm:

$g_1=b_1;$
 $g_k=(b_k+b_{k-1}) \text{ modulo } 2$; pentru $k=2, 3, \dots, n$.

Ex: $n=5$, în binar avem 101, iar în Gray avem: $1(1+0)(0+1)=111$.
Ex: $n=7$, în binar avem 111, iar în Gray avem: $1(1+1)(1+1)=100$.

! Observați faptul că soluția se construiește bit cu bit, ceea ce justifică includerea acestei probleme la tehnica Greedy.

11. Fiind dat un număr natural în cod Gray, $g_1g_2\dots g_n$, scrieți o funcție care convertește numărul în binar $b_1 b_2 \dots b_n$ după următorul algoritm:

$b_1=g_1;$
 $b_k=(g_k+b_{k-1}) \text{ modulo } 2; \text{ pentru } k=2,3\dots,n$

Ex: $n=5$, în Gray avem 111, iar în binar avem: $1(1+1)(1+0)=101$.
Ex: $n=7$, în Gray avem 100, iar în binar avem: $1(0+1)(0+1)=111$.

12. Scrieți un program care afișează codurile Gray ale primelor numere naturale citite.

13. Se consideră o mulțime X cu n elemente. Să se elaboreze un algoritm eficient și să se scrie un program pentru generarea sirului tuturor submulțimilor lui X , A_1 , A_2 , ... astfel încât A_{i+1} să se obțină din A_i , prin adăugarea sau scoaterea unui element.

14. **Problema colorării hărților.** Sunt date n țări, precizându-se relațiile de vecinătate. Se cere să se determine o posibilitate de colorare a hărții (cu cele n țări), astfel încât să nu existe țări vecine colorate la fel.

Răspunsuri / Indicații

1. a); 2. c);

3. Sortarea prin metoda bulelor are complexitatea $O(n^2)$. Apoi, selecția spectacolelor are complexitatea $O(n)$. În concluzie, complexitatea este $O(n^2)$. Dar dacă înlocuim metoda de sortare prin QuickSort, atunci complexitatea medie este $O(n \log(n))$. 4. $O(n^2)$, dar dacă se schimbă metoda de sortare, se ajunge la $O(n \log(n))$. 6. Recunoașteți problema rucsacului? În locul capacitatii rucsacului G , avem suma S , în locul greutății fiecărui obiect avem costul lui, în locul câștigului obținut din transport, avem câștigul obținut în urma vânzării produsului. 8. $O(n^2)$. 9. $O(n^3)$. 12. Se numără în baza 10, numerele se convertesc în binar, apoi în Gray. 13. De vreme ce vorbim de submulțimi, ne gândim la vectorul caracteristic. Aceasta, însă, nu va mai reține numărul în binar ci în cod Gray.

14. **Indicație.** Algoritmul propus este următorul:

- țara 1 va avea culoarea 1;
- presupunem colorate primele $i-1$ țări: țara i va fi colorată cu cea mai mică culoare (ca număr) astfel încât nici una din țările vecine să nu fie colorată la fel.

Capitolul 6

Programare dinamică

6.1. Generalități

Fie o problemă a cărei rezolvare este cerută pentru un număr natural n dat. Uneori se poate aplica un raționament de genul: dacă știm să rezolvăm problema pentru toate valorile strict mai mici decât n , atunci putem rezolva problema și pentru n dat. Dacă este așa, atunci, pe baza aceluiși raționament, înseamnă că dacă știm să rezolvăm problema pentru toate valorile strict mai mici decât $n-1$, atunci știm să rezolvăm problema pentru $n-1$ și, așa cum am arătat, pentru n . Repetând acest raționament ajungem să rezolvăm problema pentru $n=1$ și, eventual, pentru $n=2$. O astfel de problemă este foarte ușor de rezolvat. După care rezolvăm problema pentru $n=3$, apoi $n=4$, și.a.m.d., până se ajunge la acel n cerut de problemă. De aici rezultă *necesitatea găsirii unor relații de recurență*.

Ex: O persoană trebuie să urce n scări. Se știe că persoana respectivă poate urca fie o scară, fie două deodată. Întrebarea este: în câte moduri poate urca persoana n scări?

O primă idee este de a descompune pe n ca sumă de 1 și 2 în toate modurile posibile și de a număra soluțiile. Problema se poate rezolva aplicând tehnica **Backtracking** (exercițiu!). Dar în acest caz obținem un timp exponențial...

O rezolvare prin programarea dinamică se poate face în felul următor:

- dacă $n=1$, o scară se poate urca într-un singur fel;
- dacă $n=2$, se poate urca o scară și apoi o altă scară (un fel) sau deodată două scări (altă modalitate), prin urmare, există două modalități de a urca două scări.

Până acum am rezolvat problema pentru $n=1$ și pentru $n=2$. Dacă notăm cu u_n numărul de feluri în care se pot urca n scări, atunci știm că $u_1=1$ și $u_2=2$.

Acum, pentru a urca n scări putem proceda astfel: se urcă $n-2$ scări și deodată două scări sau se urcă $n-1$ scări, după care se mai urcă o scară. În câte feluri se pot urca $n-2$ scări? În u_{n-2} feluri. În câte feluri se pot urca $n-1$ scări? În u_{n-1} feluri. Atunci, n scări se pot urca în $u_n = u_{n-1} + u_{n-2}$ feluri. De ce le-am adunat? Pentru că în acest fel se obțin doar soluții diferite. Orice soluție care provine din u_{n-1} se termină prin a urca la sfârșit o scară și orice soluție care provine din u_{n-2} se termină prin a urca la sfârșit două scări. Dacă soluții sunt diferite, atunci se pot aduna.

Astfel, am obținut o relație de recurență binecunoscută. Acum, problema se reduce la a calcula u_n din această relație. Deja știm să rezolvăm o astfel de relație, revedeți recursivitatea (ați întâlnit-o la șirul lui Fibonacci). Mai mult, u_n se poate calcula în O(n), adică liniar.

Tot atunci când ați studiat recursivitatea ați văzut că rezolvarea prin utilizarea mecanismului recursivității a acestei relații este catastrofală din punct de vedere al timpului de calcul, fiind exponențială.

- În programarea dinamică, de cele mai multe ori, pentru a rezolva o problemă se scrie o relație de recurență. Relația de recurență se rezolvă apoi iterativ.

Problema prezentată face parte din categoria **problemelor de numărare**. În acest capitol veți întâlni și alte probleme de numărare.

- Prin programare dinamică puteți rezolva și **probleme de optim** în care se cere o soluție care să maximizeze sau să minimizeze o anumită funcție. În acest caz, criteriile de mai jos vă pot fi de folos. Menționăm că înțelegerea lor se face în timp, studiind mai multe exemple, motiv pentru care este bine ca, atunci când rezolvați o problemă, să reveniți asupra lor.

Se consideră o problemă în care rezultatul se obține ca urmare a unui șir de decizii D_1, D_2, \dots, D_n . În urma deciziei D_1 , sistemul evoluează din starea S_0 în starea S_1 , în urma deciziei D_2 , sistemul evoluează din starea S_1 în starea S_2 , ..., în urma deciziei D_n , sistemul evoluează din starea S_{n-1} în starea S_n .

Dacă D_1, D_2, \dots, D_n este un șir de decizii care conduce sistemul *în mod optim* din S_0 în S_n , atunci trebuie îndeplinită una din condițiile următoare (**principiul de optimalitate**):

- 1) D_k, \dots, D_n este un șir de decizii ce conduce optim sistemul din starea S_{k-1} în starea S_n , $\forall k, 1 \leq k \leq n$;
 - 2) D_1, \dots, D_k este un șir de decizii care conduce optim sistemul din starea S_0 în starea S_k , $\forall k, 1 \leq k \leq n$;
 - 3) $D_{k+1}, \dots, D_n, D_1, \dots, D_k$ sunt șiruri de decizii care conduc optim sistemul din starea S_k în starea S_n , respectiv din starea S_0 în starea S_k , $\forall k, 1 \leq k \leq n$.
- ⇒ Dacă principiul de optimalitate se verifică în forma 1), spunem că se aplică programarea dinamică **metoda înainte**.
 - ⇒ Dacă principiul de optimalitate se verifică în forma 2), spunem că se aplică programarea dinamică **metoda înapoi**.
 - ⇒ Dacă principiul de optimalitate se verifică în forma 3), spunem că se aplică programarea dinamică **metoda mixtă**.
- ! Programarea dinamică se poate aplica problemelor la care optimul general implică optimul parțial.

 Dacă drumul cel mai scurt între București și Suceava trece prin Focșani, atunci porțiunea din acest drum, dintre București și Focșani, este cea mai scurtă, ca și porțiunea dintre Focșani și Suceava (dacă n-ar fi aşa, drumul considerat între București și Suceava nu ar fi optim).

! Faptul că optimul general determină optimul parțial, nu înseamnă că optimul parțial determină optimul general.

 Fiind date drumurile cele mai scurte de la București la Cluj și de la Cluj la Suceava, nu înseamnă că drumul optim de la București la Suceava trece prin Cluj.

! Cu toate acestea, faptul că optimul general impune optimul parțial ne este de mare ajutor: căutăm optimul general, între optimele parțiale, pe care le reținem la fiecare pas. Oricum, căutarea se reduce considerabil.

6.2. Problema triunghiului

□ **Enunț.** Se consideră un triunghi de numere naturale format din n linii. Prima linie conține un număr, a doua două numere, ..., iar ultima, n numere naturale. Cu ajutorul acestui triunghi se pot forma sume de numere naturale în felul următor:

- se pornește cu numărul din linia 1;
- succesorul unui număr se află pe linia următoare plasat sub el (aceeași coloană) sau pe diagonală la dreapta (coloana crește cu 1).

Care este cea mai mare sumă care se poate forma astfel și care sunt numerele care o alcătuiesc?

Exemplu: Pentru $n=4$, se consideră triunghiul de mai jos:

2			
3	5		
6	3	4	
5	6	1	4

Se pot forma mai multe sume:

$$\begin{aligned} S_1 &= 2+3+6+5=16; \\ S_2 &= 2+5+4+1=12; \\ S_3 &= 2+3+6+6=17 \quad (\text{care este și suma maximă}). \end{aligned}$$

 **Rezolvare.** Cum am putea rezolva această problemă?

A) O primă idee ar fi să încercăm o abordare a ei prin metoda **Greedy**. Aceasta presupune că, la fiecare pas, să selectăm de pe linia respectivă cel mai mare element dintre cele două care pot fi alese. Astfel, de pe linia 1 selectăm 2, de pe linia a 2-a, 5, de pe linia a 3-a, 4, iar de pe linia a 4-a, 4. Înseamnă că suma este $2+5+4+4=15$. Observăm că în acest fel nu am reușit să obținem suma maximă. De ce? Când facem o alegere care maximizează suma la un moment dat, s-ar

putea să nu mai putem alege pentru liniile următoare elementele care maximizează suma. Urmăriți soluția optimă din exemplul dat. Prin urmare, "soluția" propusă nu este corectă.

B) Să încercăm altfel. Se cere suma maximă. Atunci ar trebui să considerăm toate "drumurile" posibile de la prima linie la ultima. Pentru fiecare astfel de drum să calculăm suma care se poate forma și să selectăm suma maximă. Este corectă o astfel de rezolvare? Evident, da. Cum am putea genera toate aceste drumuri? Dacă am spus "toate", atunci ne gândim la metoda **Backtracking**. Mai jos, prezentăm programul obținut aplicând acest algoritm:

Varianta Pascal	Varianta C++
<pre> var n,i,j,max:integer; t:array [1..50,1..50] of integer; drum,sol:array[1..50] of integer; procedure back(k:integer); var i,j,s:integer; begin s:=t[1][1]; if k=n+1 then begin for i:=2 to n do s:=s+t[i,sol[i]]; if s>max then begin for i:=1 to n do drum[i]:=sol[i]; max:=s; end end else for i:=sol[k-1] to sol[k-1]+1 do begin sol[k]:=i; back(k+1); end begin writeln('n='); readln(n); for i:=1 to n do for j:=1 to i do begin write('t[,i,',',j,']='); readln(t[i,j]); end; sol[1]:=1; back(2); writeln('Suma maxima',max); for i:=1 to n do write(drum[i],' '); end. </pre>	<pre> int t[50][50],sol[50], drum[50],n,i,j,max=0; void back(int k) { int i,j,s=t[1][1]; if (k==n+1) { for (i=2;i<=n;i++) s+=t[i][sol[i]]; if (s>max) { for (i=1;i<=n;i++) drum[i]=sol[i]; max=s; } } else { for (i=sol[k-1]; i<=sol[k-1]+1;i++) { sol[k]=i; back(k+1); } } } main() { cout<<"n="; cin>>n; for (i=1;i<=n;i++) for (j=1;j<=i;j++) { cout<<"t["<<i<<',' <<j<<"]="; cin>>t[i][j]; } sol[1]=1; back(2); cout<<"Suma maxima"<<endl; for (i=1;i<=n;i++) cout<<drum[i]<<" "; } </pre>

Să observăm că se pot forma 2^{n-1} sume de acest fel.



Exercițiu. Demonstrați prin inducție că numărul de sume care se pot forma este corect.

A lăsat în considerare toate aceste sume nu este eficient, pentru că avem un algoritm în $O(2^n)$. Am putut accepta o astfel de soluție, cu toate consecințele ei, numai dacă n-ar exista o alta, eficientă.

Exercițiu. Pentru a testa modul în care funcționează acest algoritm, modificați programul pentru a accepta intrări de la un fișier text, creați un astfel de fișier și analizați timpul în care se obține soluția pentru $n=40$.

c) Încercăm să rezolvăm problema prin aplicarea metodei programării dinamice. Verificăm principiul programării dinamice. Fie un sir de n numere care respectă condițiile problemei și care formează suma maximă: $n_1, n_2, \dots, n_i, \dots, n_n$. Este clar că numerele de la $n_1 \dots n_n$ formează o sumă maximă în raport cu sumele care se pot forma începând cu numărul n_i . Dacă această sumă nu ar fi maximă, atunci ar exista o altă alegere de numere care ar maximiza-o. Înlocuind în sirul inițial numerele de la $n_1 \dots n_n$ cu cele alese pentru a maximiza suma, vom obține o soluție în care suma este mai mare.

Pentru exemplul dat, cunoaștem soluția optimă: $2+3+6+6$. Aceasta înseamnă că, dacă de pe linia 2 se pornește cu 3, cea mai mare sumă care se poate forma, în condițiile problemei, este $3+6+6$.

Aceasta contrazice ipoteza. În această situație, se poate aplica programarea dinamică, **metoda înainte**.

Vom forma un triunghi, de la bază către vârf, cu sumele maxime care se pot forma cu fiecare număr. Dacă citim triunghiul de numere într-o matrice T și calculăm sumele într-o matrice C , vom avea relațiile următoare:

```

C[n,1]:=T[n,1];
C[n,2]:=T[n,2];
C[n,n]:=T[n,n];

```

Pentru linia i ($i < n$), cele i sume maxime se obțin astfel:

```

C[i,j]=max{T[i,j]+C[i+1,j], T[i,j]+C[i+1,j+1]},
           ie{1,2,...,n-1}, j{1,...,i}.

```

Să rezolvăm problema propusă ca exemplu.

Linia 4 a matricei C va fi linia n a matricei T :

5 6 1 4.

Linia 3 se calculează astfel:

```
C[3,1]=max{6+5, 6+6}=12;
C[3,2]=max{3+6, 3+1}=9;
C[3,3]=max{4+1, 4+4}=8;
```

Vom avea:

```
12 9 8
5 6 1 4
```

Linia 2:

```
C[2,1]=max{3+12, 3+9}=15;
C[2,2]=max{5+9, 5+8}=14;
```

```
15 14
12 9 8
5 6 1 4
```

Linia 1:

```
C[1,1]=max{2+15, 2+14}=17;
```

```
17
15 14
12 9 8
5 6 1 4
```

Aceasta este și cea mai mare sumă care se poate forma.

Pentru a tipări numerele luate în calcul se folosește o matrice numită **DRUM** în care pentru fiecare $i \in \{1, \dots, n-1\}$ și $j \in \{1, \dots, i\}$ se reține coloana în care se găsește succesorul lui $t[i, j]$.

Varianța Pascal	Varianța C++
<pre>type matrice = array [1..50,1..50] of integer; var n,i,j:integer; t,c,drum:matrice; begin write('n='); readln(n); for i:=1 to n do for j:=1 to i do begin write('t[,i,'+',j,']='); readln(t[i,j]) end; end;</pre>	<pre>#include <iostream.h> int t[50][50], c[50][50], drum[50][50], n, i, j; main() { cout<<"n="; cin>>n; for (i=1;i<=n;i++) for (j=1;j<=i;j++) { cout<<"t["<<i<<',' <<j<<"]="; cin>>t[i][j]; } }</pre>

```
for j:=1 to n do
    c[n,j]:=t[n,j];
for i:=n-1 downto 1 do
begin
    for j:=1 to i do
        if c[i+1,j]<c[i+1,j+1]
        then
            begin
                c[i,j]:=t[i,j]+
                    c[i+1,j+1];
                drum[i,j]:=j+1
            end
        else
            begin
                c[i,j]:=t[i,j]+
                    c[i+1,j];
                drum[i,j]:=j
            end
    end;
writeln('suma maxima= ',c[1,1]);
i:=1;
j:=1;
while i<=n do
begin
    writeln(t[i,j]);
    j:=drum[i,j];
    i:=i+1
end;
end.
```

```
for (j=1;j<=n;j++)
    c[n][j]=t[n][j];
for (i=n-1;i>=1;i--)
{
    for (j=1;j<=i;j++)
        if (c[i+1][j]<c[i+1][j+1])
        {
            c[i][j]=t[i][j]+
                c[i+1][j+1];
            drum[i][j]=j+1;
        }
        else
        {
            c[i][j]=t[i][j]+
                c[i+1][j];
            drum[i][j]=j;
        }
}
cout<<"suma maxima= "
    << c[1][1]<<endl;
i=1; j=1;
while (i<=n)
{
    cout<<t[i][j]<<endl;
    j=drum[i][j];
    i++;
}
```

Exerciții

- Complexitatea algoritmului pentru ultima rezolvare este $O(n^2)$. De ce?
- Puteți rezolva problema prin *metoda înapoi?*

6.3. Subșir crescător de lungime maximă

Enunț. Se consideră un vector cu n elemente întregi. Se cere să se tipărească cel mai lung subșir crescător al acestuia.

Exemplu. Pentru $n=5$ se dă vectorul $v=(4, 1, 7, 6, 7)$. În acest caz, subșirul tipărit va fi: $4, 7, 7$.

Rezolvare. Problema se poate rezolva pornind de la ideea de a calcula, pentru fiecare element al vectorului, lungimea celui mai lung subșir crescător care se poate forma începând cu el. În final, este selectat elementul din vector cu care se poate forma cel mai lung subșir crescător și acesta este listat.

$$L(k) = \{1 + \max L(i) \mid v(i) \geq v(k), i = \{k+1, \dots, n\}, k \in \{1, 2, \dots, n\}\}.$$

În practică, folosim un vector L cu n componente, unde $L(k)$ are semnificația explicită. Pentru exemplul nostru vom avea:

$$L = (3, 3, 2, 2, 1).$$

Componentele vectorului L au fost calculate astfel:

- cel mai lung subșir care se poate forma cu elementul 7, aflat pe ultima poziție, are lungimea 1;
- cel mai lung subșir care se poate forma cu elementul 6, aflat pe poziția 4, are lungimea 2 ($1+L(5)$), pentru că pe poziția 5 se găsește elementul 7 care este mai mare decât 6;
- cel mai lung subșir care se poate forma cu elementul aflat pe poziția 3 are lungimea 2 ($1+L(5)$) deoarece 7 este egal cu 7;
- algoritmul continuă în acest mod până se completează $L(1)$.

După aceasta se calculează maximul dintre componentele lui L , iar cel mai lung subșir crescător format din elementele vectorului v va avea lungimea dată de acest maxim. Pentru a lista efectiv acel subșir de lungime maximală se procedează astfel:

- se caută maximul din vectorul L precum și indicele t , la care se găsește acest maxim;
- se afișează $v(t)$;
- se găsește și se listează primul element care este mai mare sau egal cu $v(t)$ și are lungimea mai mică cu 1 ($\max - 1$), se actualizează valoarea \max cu $\max - 1$;
- algoritmul continuă până când se epuizează toate elementele subșirului.

Programul este următorul:

Varianta Pascal	Varianta C++
<pre>type vector=array[1..20] of integer; var v,l:vector; n,i,k,max,t:integer; begin write('n='); readln(n); for i:=1 to n do begin write('v['',i,'']='); readln(v[i]) end; end;</pre>	<pre>#include <iostream.h> int v[20],l[20],n,i,k,max,t; main() { cout<<"n="; cin>>n; for (i=1;i<=n;i++) { cout<<"v["<<i<<"]="; cin>>v[i]; } l[n]=1;</pre>

```

l[n]:=1;
for k:=n-1 downto 1 do
begin
  max:=0;
  for i:=k+1 to n do
    if (v[i]>=v[k]) and
       (l[i]>max)
    then max:=l[i];
  l[k]:=1+max;
end;
max:=l[1];
t:=1;
for k:=1 to n do
  if l[k]>max then
  begin
    max:=l[k];
    t:=k;
  end;
writeln('lungimea
         maxima:',max);
writeln(v[t]);
for i:=t+1 do n do
  if (v[i]>v[t]) and
     (l[i]=max-1)
  then
  begin
    writeln(v[i]);
    max:=max-1
  end
end.

```

```

for (k=n-1;k>=1;k--)
{
  max=0;
  for (i=k+1;i<=n;i++)
    if (v[i]>=v[k] && l[i]>max)
      max=l[i];
  l[k]=1+max;
}
max=l[1];
t=1;
for (k=1;k<=n;k++)
if (l[k]>max)
  { max=l[k];
    t=k;
  }
cout<<"lungimea maxima:"<<max
     <<endl<<v[t]<<endl;
for (i=t+1;i<=n;i++)
if (v[i]>v[t] && l[i]==max-1)
{ cout<<v[i]<<endl;
  max--;
}
}

```



Exercițiu. Cum verificăți pentru această problemă principiul optimalității?

Complexitatea acestui algoritm este $O(n^2)$.

Pentru fiecare element al sirului se calculează un maxim. Aceasta presupune parcurgerea sirului până la capăt. Pentru elementul aflat pe poziția $n-1$ se face o comparare, pentru elementul aflat pe poziția $n-2$ se fac 2 comparații, ..., pentru elementul aflat pe poziția 1 se fac $n-1$ comparații.

Prin urmare, numărul de comparații este:

$$S = 1 + 2 + \dots + n - 1 = \frac{n(n-1)}{2},$$

deci algoritmul are complexitatea $O(n^2)$.

6.4. O problemă cu sume

Enunț. Se citește $n > 1$, număr natural, **Suma**, număr natural și n numere naturale nenule. Se cere să se decidă dacă numărul **Suma** poate fi obținut ca sumă de numere naturale dintre cele n citite. În caz afirmativ, să se afișeze un set de numere care, adunate, dau acest rezultat.

Rezolvare. Putem considera toate submulțimile mulțimii $\{1, 2, \dots, n\}$ și pentru fiecare submulțime calculăm suma obținută. Dar... câte submulțimi avem? Avem 2^n submulțimi. Înseamnă că avem un algoritm în $O(2^n)$... Inaceptabil!

Vom prefera un alt algoritm, bazat pe metoda programării dinamice.

Notăm numerele citite cu n_1, n_2, \dots, n_n . Fie $S = n_1 + n_2 + \dots + n_n$. Evident, orice sumă care se poate forma cu numerele citite, poate fi un număr între 1 și S . Vectorul **Suma**, un vector cu S componente, va reține 1 pentru fiecare sumă care poate fi formată și 0 în caz contrar.

La pasul 1 vom calcula toate sumele care se pot calcula cu numerele n_1, n_2, \dots, n_n .

La pasul 1 avem **Sume** [n_1] = 1.

La pasul 2 avem **Sume** [n_2] = 1 și **Sume** [$n_1 + n_2$] = 1.

La pasul 3 avem **Sume** [n_3] = 1 și **Sume** [$n_1 + n_2 + n_3$] = 1, **Sume** [$n_1 + n_2 + n_3$] = 1.

...

Dar cum reținem toți termenii care alcătuiesc o sumă? Mai simplu decât pare la prima vedere... Un vector, numit **Alege**, cu S componente, va reține pentru fiecare sumă calculată **ultimul termen** care intră în alcătuirea ei. Atunci când afișăm soluția pentru **Suma**, tipărim **Alege** [**Suma**], apoi **Alege** [**Suma**]-**Alege** [**Suma**]...

Ex: Vom prezenta ca exemplu cazul $n=3$. Numerele citite sunt 2, 3, 5.

Sumele calculate pot lua valori între 1 și $2+3+5=10$. Un vector **Suma**, cu 10 componente va reține 1 pentru fiecare sumă care se poate calcula și 0 dacă, până la acel pas suma nu s-a putut calcula.

Cu numărul 2 se poate forma o singură sumă: 2.

Sume	Alege																																								
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	0	1	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	0	2	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0																																
1	2	3	4	5	6	7	8	9	10																																
0	2	0	0	0	0	0	0	0	0																																
1	2	3	4	5	6	7	8	9	10																																

Cu numerele 2 și 3 se mai pot forma sumele: 3, și $5=2+3$.

Suma	Alege																																								
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	0	1	1	0	1	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	0	2	3	0	3	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10
0	1	1	0	1	0	0	0	0	0																																
1	2	3	4	5	6	7	8	9	10																																
0	2	3	0	3	0	0	0	0	0																																
1	2	3	4	5	6	7	8	9	10																																

Cu numerele 2, 3, 5 se mai pot forma și sumele: 5 (a mai fost obținută), $7=2+5$ și $8=3+5$, $10=5+5$.

Suma	Alege																																								
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	0	1	1	0	1	0	1	1	0	1	1	2	3	4	5	6	7	8	9	10	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>0</td><td>2</td><td>3</td><td>0</td><td>5</td><td>0</td><td>5</td><td>5</td><td>0</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> </table>	0	2	3	0	5	0	5	5	0	5	1	2	3	4	5	6	7	8	9	10
0	1	1	0	1	0	1	1	0	1																																
1	2	3	4	5	6	7	8	9	10																																
0	2	3	0	5	0	5	5	0	5																																
1	2	3	4	5	6	7	8	9	10																																

Dacă **Suma**=7, se afișează **Alege** [7]=5, **Alege** [7-5]= **Alege** [2]=2.

Complexitatea algoritmului. Fie s suma numerelor. Algoritmul tratează fiecare număr citit (n numere) și pentru fiecare număr citit parcurge vectorul **Suma** care are s componente. Prin urmare, complexitatea lui este $O(ns)$.

! Pentru a nu calcula o sumă de mai multe ori, pentru calculul noilor sume se utilizează un vector auxiliar, numit **Sume1**, după care se execută sau logic între conținuturile celor doi vectori (vedeți programul următor).

Varianța Pascal	Varianța C++
<pre>var Nr,Sume,Sume1,Alege:array [0..100] of integer; n,i,j,S,Suma:integer; begin write ('n='); readln(n); for i:=1 to n do begin readln(Nr[i]); S:=S+Nr[i] end; for i:=1 to n do begin Sume1[Nr[i]]:=1; for j:=1 to S do if (Sume1[j]=1) then Sume1[j+Nr[i]]:=1; for j:=1 to S do if (Sume1[j]=1) and (Sume1[j]=0) then begin Sume1[j]:=1; Alege[j]:=Nr[i]; Sume1[j]:=0; end; end; end;</pre>	<pre>#include <iostream.h> int Nr[100],Sume[100], Sume1[100],Alege[100],n,i,j, S,Suma; main() { cout<<"n="; cin>>n; for(i=1;i<=n;i++) { cin>>Nr[i]; S+=Nr[i]; } for (i=1;i<=n;i++) { Sume1[Nr[i]]=1; for (j=1;j<=S;j++) if(Sume1[j]==1) Sume1[j+Nr[i]]=1; for(j=1;j<=S;j++) if (Sume1[j]==1) and (Sume1[j]==0) then { Sume1[j]=1; Alege[j]=Nr[i]; Sume1[j]=0; } } }</pre>

```

write('Suma='); readln(Suma);
if Suma<=S then
  if Alege[Suma]<>0 then
    while Suma<>0 do
      begin
        writeln(Alege[Suma]);
        Suma:=Suma-Alege[Suma];
      end
    else writeln ('Suma nu se
                  poate forma ')
  else writeln ('Suma nu se
                  poate forma ')
end.

```

```

cout<<"Suma="; cin>>Suma;
if (Suma<=S)
  if (Alege[Suma])
    while (Suma)
    { cout<<Alege[Suma]<<endl;
      Suma-=Alege[Suma];
    }
  else cout<<"Suma nu se poate forma";
  else cout<<"Suma nu se poate forma";
}

```

! Dacă numerele sunt introduse în ordine crescătoare, atunci suma se obține prin utilizarea unui număr maxim de termeni și invers, dacă numerele sunt introduse în ordine descrescătoare, atunci suma se obține prin utilizarea unui număr minim de termeni. De ce?

6.5. Problema rucsacului (cazul discret)

Enunț (Varianta 1). O persoană are la dispoziție un rucsac cu o capacitate de G unități de greutate și intenționează să efectueze un transport în urma căruia să obțină un câștig. Persoana are la dispoziție n obiecte. Pentru fiecare obiect se cunoaște greutatea sa Gr_i (număr natural) și câștigul obținut în urma transportului său: c_i . Ce obiecte trebuie să aleagă persoana pentru a-și maximiza câștigul și care este acesta?

Cerință suplimentară. Odată ales un obiect, acesta trebuie transportat integral (nu se admite transportul unei părți din el). În acest fel trebuie să rezolvăm **problema discretă a rucsacului (rucsac 0/1)**.

Rezolvare. Vom nota obiectele cu $1, 2, \dots, n$. Există posibilitatea ca problema să admită mai multe soluții optime. Fie $S=\{i_1, i_2, \dots, i_k\}$ o soluție optimă a problemei (unde $i_1 < i_2 < \dots < i_k$ sunt obiecte dintre cele n). Dacă se înlătură obiectul i_k , atunci greutatea $G-Gr(i_k)$ este încărcată optim cu obiectele $i_1 < i_2 < \dots < i_{k-1}$. Dacă, prin absurd, ar exista o altă încărcare a rucsacului pentru greutatea $G-Gr(i_k)$, care aduce un câștig mai mare, atunci, dacă la acea încărcătură s-ar adăuga obiectul i_k , am obține o soluție mai bună decât cea inițială, ceea ce contrazice optimalitatea soluției.

Aceasta conduce la următoarea idee de rezolvare a problemei:

- Capacitățile $1, 2, \dots, G$ se încarcă optim, la început cu obiectul 1, apoi se îmbunătățește soluția cu obiectul 2, ... și, la sfârșit, se îmbunătățește soluția cu obiectul n .

- Pentru a calcula câștigul maxim vom utiliza matricea $Castig_{n+1, G+1}$, unde $Castig(i, j)$ va reține câștigul obținut prin transportul obiectelor $1, 2, \dots, i$, dacă capacitatea de transport este j . Relațiile de recurență sunt următoarele:
 - $Castig(i, 0) = 0, \quad i = 1 \dots n$
 - $Castig(0, j) = 0, \quad j = 1 \dots G$
 - $Castig(i, j) = \begin{cases} Castig(i-1, j - Gr(i)) + C(i), & \text{dacă } Castig(i-1, j - Gr(i)) + C(i) > Castig(i-1, j) \\ Castig(i-1, j), & \text{altfel} \end{cases}$

Justificarea relațiilor

- Dacă rucsacul are capacitatea 0, evident, nu poate fi transportat nici un obiect, în concluzie, în toate cazurile câștigul este 0.
- Dacă nu transportăm nici un obiect, atunci, indiferent de capacitate, câștigul obținut este 0.
- Cazul corespunde deciziei de a alege sau nu pentru greutatea j produsul i . Dacă pentru greutatea j , se alege produsul i , atunci câștigul care se obține este suma dintre câștigul maxim obținut pentru capacitatea $j - Gr(i)$ la care se adaugă câștigul obținut din transportul obiectului i . Dacă nu se alege spre transport obiectul i , atunci ne mulțumim cu câștigul maxim obținut pentru greutatea j , în cazul în care se transportă doar obiecte alese dintre primele $i-1$. Evident, alegem sau nu pentru transport obiectul i în funcție de câștigul care se obține, care trebuie să fie maxim.

Să observăm că matricea $Castig$ se completează pe linii.

Exemplu: Pentru $n=5$:

Obiect	1	2	3	4	5
greutate	3	4	3	10	5
castig	2	3	8	12	7

Capacitatea rucsacului este de 10 unități de greutate. Persoana va alege obiectele 3 și 5 și va obține un câștig de 15 unități monetare. În acest fel se vor transporta numai 8 unități de greutate.

Îată cum arată matricea $Castig$ și matricea $Alege$ după actualizarea, pe rând, cu obiectele 1, 2, ..., n :

0	0	2	2	2	2	2	2	2	2
0	0	2	3	3	3	5	5	5	5
0	0	8	8	8	10	11	11	11	13
0	0	8	8	8	10	11	11	11	13
0	0	8	8	8	10	11	15	15	15

matricea $Castig$

0	0	1	1	1	1	1	1	1	1
0	0	1	2	2	2	2	2	2	2
0	0	3	3	3	3	3	3	3	3
0	0	3	3	3	3	3	3	3	3
0	0	3	3	3	3	3	3	3	3

matricea $Alege$

Varianța Pascal	Varianța C++
<pre> var Castig,alege:array[0..50, 0..50] of integer; Gr,C:array [1..100] of integer; i,j,n,G,Obiect:integer; begin write ('G='); readln(G); write ('n='); readln(n); for i:=1 to n do begin write ('Gr[,i,]'='); readln(Gr[i]); write ('C[,i,]'='); readln(C[i]); end; for i:=1 to n do for j:=1 to G do if Gr[i]<=j then if C[i]+Castig[i-1, j-Gr[i]]>Castig[i-1,j] then begin Castig[i,j]:=C[i]+ Castig[i-1,j-Gr[i]]; Alege[i,j]:=i; end else begin Castig[i,j]:=- Castig[i-1,j]; Alege[i,j]:=-Alege[i-1,j]; end else begin Castig[i,j]:=Castig[i-1,j]; Alege[i,j]:=Alege[i-1,j]; end; end; i:=n; j:=G; writeln('Castig total= ', Castig[i,j]); while Alege[i,j]<>0 do begin Obiect:=Alege[i,j]; writeln (' Produsul ', Alege[i,j], ' greutate ', Gr[Alege[i,j]], ' castig ', C[Alege[i,j]]); while Obiect=Alege[i,j] do begin j:= j-Gr[Alege[i,j]]; i:=i-1; end; end; end. </pre>	<pre> #include <iostream.h> int Castig[50][50], Alege[50][50],Gr[100],C[100], i,j,n,G,Obiect; main() { cout<<"G=";cin>>G; cout<<"n=";cin>>n; for(i=1;i<=n;i++) { cout<<"Gr["<<i<<"]="; cin>>Gr[i]; cout<<"C["<<i<<"]="; cin>>C[i]; } for(i=1;i<=n;i++) for(j=1;j<=G;j++) if (Gr[i]<=j) if(C[i]+Castig[i-1][j-Gr[i]]>Castig[i-1][j]) Castig[i][j]=C[i]+ Castig[i-1][j-Gr[i]]; Alege[i][j]=i; else (Castig[i][j]=Castig[i-1][j], Alege[i][j]=Alege[i-1][j]); else (Castig[i][j]=Castig[i-1][j], Alege[i][j]=Alege[i-1][j]); i=n; j=G; cout<<"Castig total " <<Castig[i][j]<<endl; while (Alege[i][j]) { Obiect=Alege[i][j]; cout<<" Produsul " <<Alege[i][j] <<" Greutate " <<Gr[Alege[i][j]] <<" Castig " <<C[Alege[i][j]]<<endl; while (Obiect==Alege[i][j]) { j-=Alege[i][j]; i--; } } } </pre>

Enunț (Varianță 2). La fel ca la prima variantă, numai că se presupune că se dispune de un număr nelimitat de obiecte de un anumit tip. Pentru fiecare tip de obiect se cunoaște greutatea unui exemplar și câștigul obținut prin transportul său la destinație.

Rezolvare. Pare mai complicat, dar, în realitate, este algoritmul de la problema anterioară, simplificat. Matricele **Castig** și **Alege** devin vectori cu **G+1** componente. Se parcurg vectorii de mai sus, o dată pentru tipul de obiect 1, apoi pentru tipul de obiect 2, ... și, la sfârșit, pentru tipul de obiect **n**. La fiecare parcurgere se urmărește să se încarcă optim greutățile **1,2,...,G**.

Spre deosebire de algoritmul anterior, unde, la pasul **i**, se actualizează încărcarea greutății **j**, pornind de la greutatea **j-G[i]**, încărcată optim cu obiecte de tipuri **1, 2, ..., i-1**, aici se actualizează greutatea **j-G[i]**, încărcată optim cu produsele **1, 2, ..., i-1, i**. Aceasta face să se selecteze mai multe produse de același tip. Evident, există posibilitatea să obtinem un câștig general mai mare decât la problema precedentă.

Pentru exemplul anterior, puteți observa mai jos conținuturile vectorilor **Castig** și **Alege**, după rulare. Soluția aleasă va fi: **Castig 24**, se alege de 3 ori obiectul 3.

```

1 castig 0 alege 0
2 castig 0 alege 0
3 castig 8 alege 3
4 castig 8 alege 3
5 castig 8 alege 3
6 castig 16 alege 3
7 castig 16 alege 3
8 castig 16 alege 3
9 castig 24 alege 3

```

Programul este prezentat mai jos:

Varianța Pascal	Varianța C++
<pre> var Castig,Alege:array[0..100] of integer; Gr,C:array [1..100] of integer; i,j,n,G:integer; begin write ('G='); readln(G); write ('n='); readln(n); for i:=1 to n do begin write ('Gr[,i,]'='); readln(Gr[i]); write ('C[,i,]'='); readln(C[i]); end; end; </pre>	<pre> #include <iostream.h> int Castig[100],Alege[100], Gr[100],C[100],i,j,n,G; main() { cout<<"G="; cin>>G; cout<<"n="; cin>>n; for(i=1;i<=n;i++) { cout<<"Gr["<<i<<"]="; cin>>Gr[i]; cout<<"C["<<i<<"]="; cin>>C[i]; } } </pre>

```

for i:=1 to n do
  for j:=1 to G do
    if Gr[i]<=j then
      if C[i]+Castig[j-Gr[i]] >
        Castig[j] then
      begin
        Castig[j]:=C[i]+
          Castig[j-Gr[i]];
        Alege[j]:=i;
      end;
      writeln('Castig total=',
        Castig[G]);
    j:=G;
    while alege[j]<>0 do
    begin
      writeln('obiectul ',
        alege[j], 'castig ',
        C[alege[j]]);
      j:=j-Gr[alege[j]];
    end
  end.
}

for(i=1;i<=n;i++)
  for(j=1;j<=G;j++)
    if (Gr[i]<=j)
      if(C[i]+Castig[j-Gr[i]] >
        Castig[j])
      { Castig[j]=C[i]+
        Castig[j-Gr[i]];
        Alege[j]=i;
      }
    j=G;
    cout<<"Castig total "
    <<Castig[G]<<endl;
    while (Alege[j])
    { cout<<" Produsul "
      <<Alege[j]
      <<" Greutate "
      <<Gr[Alege[j]]
      <<" Castig "
      <<C[Alege[j]]<<endl;
      j-=Gr[Alege[j]];
    }
}

```

- ✓ Indiferent de variantă, complexitatea este $O(nG)$.
- ✓ O astfel de complexitate se numește complexitate pseudopolinomială. În unele cazuri se obțin soluții într-un timp foarte scurt, dar... ce ne facem dacă G este foarte mare, de exemplu $G=2^n$.

Mai jos, puteți analiza **modelul matematic** al problemei rucsacului, varianta discretă, cazul 1. Puteți interpreta relațiile?

Se cer $X_1, X_2, \dots, X_n \in \{0,1\}$, astfel încât:

$$\begin{cases} g_1X_1 + g_2X_2 + g_3X_3 + \dots + g_nX_n \leq G & g_1, g_2, \dots, g_n, G \in \mathbb{N} \\ \max f = c_1X_1 + c_2X_2 + c_3X_3 + \dots + c_nX_n, & c_1, c_2, \dots, c_n \in \mathbb{N} \end{cases}$$

Chiar dacă matematicienilor le place, uneori, să prezinte unele probleme pe un exemplu copilăresc, ca în cazul de față, unde o persoană are un rucsac și încearcă să transporte niște obiecte, problema are o importanță uriașă în economie.

Analizați exemplele următoare:

- 1) O firmă de transport dispune de un vapor și într-un port găsește mai multe produse pe care le poate transporta în țara de origine. În urma transportului se pot obține anumite câștiguri care se cunosc de la început, pentru fiecare produs în parte. Iată că în locul rucsacului avem un vapor, deși problema prezintă interes... economic!

Exerciții

1. Cum particularizați problema pentru cazul continuu al problemei rucsacului (revedeți Greedy)?
 2. Cum particularizați problema pentru cazul discret forma 1?
 3. Cum particularizați problema pentru cazul discret forma 2?
- 2) O firmă poate fabrica mai multe produse: p_1, p_2, \dots, p_n . În acest scop firma dispune de un o sumă de bani, s . Pentru fiecare produs se cunoaște costul fabricării și beneficiul obținut în urma vânzării lui. Ce produse trebuie să fabrice firma pentru ca beneficiul obținut să fie maxim?

Problema se rezolvă imediat pornind de la modelul matematic al problemei rucsacului. Ce semnificație au în acest caz constantele din model?

- 3) O firmă dispune de o sumă de bani s , în vederea investirii ei. Se pot face n investiții și pentru fiecare investiție în parte se cunoaște suma necesară. De asemenea, se știe că fiecare investiție aduce, după un număr de ani, un câștig anual. Ce investiții trebuie să facă firma astfel încât să-și maximizeze beneficiile anuale? și această problemă se rezolvă imediat pornind de la modelul matematic al problemei rucsacului. Ce semnificație au în acest caz constantele din model?

6.6. Distanța Levenshtein

Enunț. Se consideră două cuvinte A și B cu m , respectiv, n caractere. Se cere să se transforme cuvântul A în cuvântul B prin utilizarea a 3 operații:

A – adăugarea unei litere;

M – modificarea unei litere;

S – ștergerea unei litere.

Transformarea se va face prin utilizarea unui număr minim de operații. Se va afișa numărul minim de operații și sirul transformărilor.

Exemplu: $A = 'IOANA'$ ($m=5$), $B = 'DANIA'$ ($n=5$).

$$IOANA \xrightarrow{S} OANA \xrightarrow{M} DAN \xrightarrow{A} DANIA$$

Se va afișa 3 (numărul transformărilor).

! Numărul minim de transformări se numește **distanța Levenshtein** (notiunea a fost introdusă, împreună cu algoritmul de calcul, în anul 1965 de către omul de știință rus Vladimir Levenshtein).

Rezolvare. Să observăm că numărul de operații necesar conversiei este mai mic sau egal cu maximul dintre m și n . De exemplu, dacă $m < n$, putem modifica primele m caractere și șterge ultimele $n-m$ caractere.

Fie sirul optim care transformă pe A în B . Atunci, sirul transformărilor de la A la T_k este optim. Dacă, prin absurd, nu ar fi optim, înseamnă că există un alt sir cu număr mai mic de transformări de la A la T_k . Dacă înlocuim acest sir în sirul transformărilor de la A la B , se obțin mai puține transformări, deci se contrazice optimalitatea soluției inițiale. În concluzie, sirul transformărilor de la A la T_k este optim.

$$A \rightarrow T_1 \rightarrow T_2 \rightarrow \dots T_k \rightarrow \dots B$$

În sirul transformărilor, de la un termen la altul, se trece prin efectuarea unei singure operații dintre: adăugarea, modificarea sau ștergerea unui caracter. Aceasta înseamnă că doi termeni consecutivi ai sirului diferă printr-un singur caracter. Mai mult, în sirul transformărilor, nu contează ordinea în care efectuăm calculele.

De exemplu, un sir optim de transformări de la **IOANA** la **DANIA** este și:

$$\text{IOANA} \xrightarrow{A} \text{IOANIA} \xrightarrow{s} \text{OANIA} \xrightarrow{M} \text{DANIA}$$

pe lângă sirul modificărilor de mai jos:

$$\text{IOANA} \xrightarrow{s} \text{OANA} \xrightarrow{M} \text{DANA} \xrightarrow{A} \text{DANIA}$$

În ambele siruri s-au modificat, adăugat, șters aceeași caracter, aflate pe aceeași poziție în *sirul inițial*, doar ordinea diferă. Astfel, în primul sir am adăugat caracterul **I** pe poziția 4, am șters caracterul **I** de pe prima poziție, iar caracterul **O** de pe poziția 2 a fost modificat în **D**. În al doilea sir, am șters caracterul **I** de pe prima poziție, am modificat caracterul **O** de pe a doua poziție în **D** și am adăugat caracterul **I** pe poziția 4.

Dacă ordinea transformărilor nu contează, atunci vom prefera să calculăm numărul minim al transformărilor, pornind de la A , și numărând adăugările, modificările, și ștergerile caracterelor de la stânga către dreapta (în *sirul inițial*).

Pentru rezolvare, vom presupune că fiecare cuvânt (A , B) începe cu un caracter vid, pe care-l notăm cu v . În aceste condiții, avem de efectuat transformarea:

$$vA_1 A_2 \dots A_m \rightarrow vB_1 B_2 \dots B_n$$

În acest fel, A va avea $m+1$ caractere, iar B va avea $n+1$ caractere.

Pentru calculul distanței vom utiliza matricea Cost cu $m+1$ linii și $n+1$ coloane. Linile sunt între 0 și $m+1$, iar coloanele între 0 și $n+1$. Elementul $\text{Cost}[i, j]$ va reține numărul minim al transformărilor primelor i caractere ale cuvântului A în primele j caractere ale cuvântului B . Mai precis, $\text{Cost}[i, j]$ înseamnă costul obținerii cuvântului intermedian:

$$B_1 B_2 \dots B_j A_{j+1} A_{j+2} \dots A_m$$

Avem:

1) $\text{Cost}[0, j] = j$, $j=0, 1, 2, \dots, n$. Semnificație: costul transformării caracterului vid, care precede pe A , în primele j caractere ale cuvântului B , este j (se fac j adăugări).

2) $\text{Cost}[i, 0] = i$, $i=0, 1, 2, \dots, m$. Semnificație: costul transformării primelor i caractere ale lui A în caracterul vid care îl precede pe B , este i (se fac i ștergeri).

3) $0 < i \leq m$, $0 < j \leq n$ +

$$\text{Cost}[i, j] = \begin{cases} \text{Cost}[i-1, j-1], & A_i = B_j \\ 1 + \min\{\text{cost}(i-1, j-1), \text{cost}(i, j-1), \text{cost}(i-1, j)\}, & \text{altfel} \end{cases}$$

Semnificație: la fiecare pas, se compară caracterul aflat în A pe poziția i ($A[i]$), cu cel aflat în B pe poziția j ($B[j]$). Dacă se cunosc costurile optime (numărul minim de operații): $C[i-1, j-1]$, $C[i, j-1]$, $C[i-1, j]$, atunci:

a) Dacă caracterul aflat pe poziția i în A este egal cu caracterul aflat pe poziția j în B ($A_i = B_j$), se șterge acel caracter. În acest caz,

$$\text{Cost}[i, j] = \text{Cost}[i-1, j-1].$$

$$B_1 B_2 \dots B_{j-1} A_i A_{i+1} \dots A_m \rightarrow B_1 B_2 \dots B_{j-1} B_j A_{i+1} \dots A_m$$

b) $A_i \neq B_j$. Atunci se efectuează una din operațiile următoare, mai precis, cea care are costul cel mai mic:

b1) Adăugarea unui caracter (B_j). Atunci costul total este: $1 + \text{Cost}[i, j-1]$.

$$B_1 B_2 \dots B_{j-1} A_i A_{i+1} \dots A_m \rightarrow B_1 B_2 \dots B_{j-1} B_j A_{i+1} \dots A_m$$

b2) Ștergerea unui caracter, cel aflat pe poziția A_i . Atunci costul total este $1 + \text{Cost}[i-1, j]$.

$$B_1 B_2 \dots B_{j-1} B_j A_{i+1} \dots A_m \rightarrow B_1 B_2 \dots B_{j-1} A_i A_{i+1} \dots A_m$$

b3) Modificarea unui caracter, A_i va fi egal B_j . Atunci costul total este $1 + \text{Cost}[i-1, j-1]$.

$$B_1 B_2 \dots B_{j-1} A_i A_{i+1} \dots A_m \rightarrow B_1 B_2 \dots B_{j-1} B_j A_{i+1} \dots A_m$$

Ex: Cazul "IOANA→DANIA". Costul transformării caracterului vid în caracterul vid este 0, costul transformării caracterului vid în "D" este 1, costul transformării caracterului vid în "DA" este 2, ..., costul transformării caracterului vid în "DANIA" este 5. Apoi, costul transformării caracterului vid în caracterul vid este 0, costul transformării caracterului "I" în caracterul vid este 1, costul transformării caracterelor "IO" în caracterul vid este 2, ..., costul transformării caracterelor "IOANA" în caracterul vid este 5.

```
VDANIA
V012345
I1
O2
A3
N4
A5
```

În continuare, completăm matricea pe linii:

$C[1,1]. A[1]=I \neq D=B[1]$.

$1+\min\{cost[i-1,j-1], cost[i-1,j], cost[i,j-1]\}=1+\min\{cost[0,0], cost[0,1], cost[1,0]\}=1+\min\{0,1,1\}=1$.

Semnificația: modificarea minimă pentru a transforma "I" în "D" are costul 1. Se face o modificare pentru că $(i-1, i-1)$ trece în (i, j) .

```
VDANIA
V012345
I11
O2
A3
N4
A5
...
...
```

În final, matricea este:

```
VDANIA
V012345
I112334
O222344
A332344
N443234
A554333
```

Numărul minim de transformări este: $cost[m,n]=cost[5,5]=3$.

După calculul elementelor matricei, se depistează operațiile efectuate și acestea se afișează în ordine inversă.

Depistarea unei operații

Pentru $i=5$ și $j=5$,

$$\min\{cost[4,4], cost[4,5], cost[5,4]\} = \min\{3, 4, 3\}$$

Alegem 3 (prima valoare minimă din sir). Pentru că avem $cost[5,5]=\min$, înseamnă că la ultima operație s-a efectuat un salt (a fost găsită egalitate).

Acum $i=4$ și $j=4$. Apoi:

$$\min\{cost[3,3], cost[3,4], cost[4,3]\} = \min\{3, 4, 2\} = 2$$

$$cost[4,4] \neq \min$$

Pentru că am efectuat o operație de tipul $(i, j-1) \rightarrow (i, j)$ înseamnă că s-a adăugat pe poziția 4 a sirului A caracterul de pe poziția 4 a sirului B (i). Aceasta este ultima modificare făcută de algoritm.

Avem $i=4$, $j=3$. Depistăm apoi, penultima modificare, să.m.d., până când $i=0$ și $j=0$. Pentru a afișa modificările în ordine inversă vom utiliza o stivă (Sol). De asemenea, se poate utiliza recursitatea.

Varianta Pascal	Varianta C++
<pre>var A,B:string; Sol:array[1..100] of string; m,n,i,j,min,k:integer; op:char; cost:array[0..100,0..100] of integer; procedure Next(var i,j: integer;var op:char); var l,c:integer; begin l:=0;c:=0; min:=1000; if (i>0) and (j>0) and (min>cost[i-1,j-1]) then begin min:=cost[i-1,j-1]; l:=i-1; c:=j-1; op:='m'; end; if (i>0) and (cost[i-1,j]<min) then begin min:=cost[i-1,j]; l:=i-1;c:=j; op:='s'; end; if (j>0) and (cost[i,j-1]<min) then begin min:=cost[i,j-1]; l:=i;c:=j-1; op:='a'; end; if (cost[i][j]==min) op='v'; i:=l; j:=c; end;</pre>	<pre>#include <iostream.h> #include <string.h> char A[100],B[100], Sol[100][300],op; int m,n,i,j,min,k, cost[100][100]; void Next(int& i, int& j, char& op) { int l=0,c=0; min=1000; if (i>0 && j>0 && min>=cost[i-1][j-1]) { min=cost[i-1][j-1]; l=i-1; c=j-1; op='m'; } if (i>0 && cost[i-1][j]<min) { min=cost[i-1][j]; l=i-1; c=j; op='s'; } if (j>0 && cost[i][j-1]<min) { min=cost[i][j-1]; l=i; c=j-1; op='a'; } if (cost[i][j]==min) op='v'; i=l; j=c; }</pre>

```

if (j>0) and (cost[i,j-1]<min)
then
begin
  min:=cost[i,j-1];
  l:=i;c:=j-1;
  op:='a';
end;
if cost[i,j]=min then op:='v';
i:=l;j:=c;
end;
begin
  write('A='); readln(A);
  write('B='); readln(B);
  m:=length(A);
  n:=length(B);
  for i:=0 to m do
    cost[i,0]:=i;
  for j:=0 to n do
    cost[0,j]:=j;
for i:=1 to m do
  for j:=1 to n do
    if A[i]=B[j]
    then
      cost[i,j]:=cost[i-1,j-1]
    else
      begin
        min:=cost[i-1,j-1];
        if min>cost[i-1,j] then
          min:=cost[i-1,j];
        if min>cost[i,j-1] then
          min:=cost[i,j-1];
        cost[i,j]:=i+min;
      end;
writeln ('Distanta=',
         cost[m,n]);
i:=m; j:=n;
while i+j>0 do
begin
  Next(i,j,op);
  if op<>'v'
  then
    begin
      k:=k+1;
      Sol[k]:=op+
        '+copy(b,1,j)+
        copy(a,i+1,m-i);
    end;
  for k:= cost[m,n] downto 1 do
    writeln(Sol[k]);
  writeln(B);
end.

```

După cum se poate observa, algoritmul are complexitatea $O(n^2)$.

```

main()
{ int t;
cout<<"A="; cin>>A+1;
cout<<"B="; cin>>B+1;
m=strlen(A+1);
n=strlen(B+1);
cout<<m<<" "<<n<<endl;
for (i=0;i<=m;i++)
  cost[i][0]=i;
for(j=0;j<=n;j++)
  cost[0][j]=j;
for (i=1;i<=m;i++)
  for(j=1;j<=n;j++)
    if(A[i]==B[j])
      cost[i][j]=cost[i-1][j-1];
    else
      { min=cost[i-1][j-1];
        if (min>cost[i-1][j])
          min=cost[i-1][j];
        if (min>cost[i][j-1])
          min=cost[i][j-1];
        cost[i][j]=1+min;
      }
    cout<<"Distanta "
      <<cost[m][n]<<endl;
  i=m;
  j=n;
  while(i+j)
  { Next(i,j,op);
    if (op!= 'v')
    { int x=0;
      k++;
      Sol[k][x++]=op;
      Sol[k][x++]=' ';
      for (t=1;t<=j;t++)
        Sol[k][x++]=B[t];
      for (t=i+1;t<=m;t++)
        Sol[k][x++]=A[t];
    }
    for(k=cost[m][n];k>=1;k--)
      cout<<Sol[k]<<endl;
    cout<<B+1;
  }
}

```

✓ **Aplicație.** Un profesor de informatică dorește să-și dea seama în ce măsură, la o lucrare în care elevilor li s-a cerut să elaboreze un program, aceștia s-au "inspirat" unul de la altul. Profesorul a pornit de la observația că elevii "spațiază" altfel programul și schimbă numele unor variabile. Ideea care i-a venit constă în a prelua programul, eliminând toate blank-urile și a forma astfel, din tot programul, un sir de caractere. La fel procedează și cu alt program. În final, calculează **distanța Levenshtein** între cele două programe. Evident, cu cât costul transformării unui sir (program) în altul este mai mic, cu atât suspiciunea de inspirație este mai mare. Puteți să scrieți programul care realizează această prelucrare? Programele le găsiți în fișiere text, cu extensia .pas sau .cpp, aşa cum le scrieți în mod obișnuit!

6.7. Înmulțirea optimă a unui sir de matrice

Presupunem că avem de înmulțit două matrice: $A_{n,p}$ cu $B_{p,m}$. În mod evident, rezultatul va fi o matrice $C_{n,m}$. Se pune problema de a afla câte înmulțiri au fost făcute pentru a obține matricea C . Prin înmulțirea liniei 1 cu coloana 1 se fac p înmulțiri, întrucât au p elemente. Dar linia 1 se înmulțește cu toate cele m coloane, deci se fac $m \cdot p$ înmulțiri. În mod analog se procedează cu toate cele n linii ale matricei A , deci se fac $n \cdot m \cdot p$ înmulțiri. Reținem acest rezultat.

E: Să considerăm produsul de matrice $A_1 \times A_2 \times \dots \times A_n$ ($A_1(d_1, d_2)$, $A_2(d_2, d_3), \dots, A_n(d_n, d_{n+1})$). Se cunoaște că legea de compozitie produs de matrice nu este comutativă, în schimb este asociativă. De exemplu, dacă avem de înmulțit trei matrice A, B, C produsul se poate face în două moduri: $(AxB)xC$ sau $AxBxC$.

Este interesant de observat că nu este indiferent modul de înmulțire a celor n matrice. Să considerăm că avem de înmulțit patru matrice $A_1(10, 1)$, $A_2(1, 10)$, $A_3(10, 1)$, $A_4(1, 10)$.

Pentru înmulțirea lui A_1 cu A_2 se fac 100 de înmulțiri și se obține o matrice cu 10 linii și 10 coloane. Prin înmulțirea acesteia cu A_3 se fac 100 de înmulțiri și se obține o matrice cu 10 linii și o coloană. Dacă această matrice se înmulțește cu A_4 , se fac 100 de înmulțiri. În concluzie, dacă acest produs se efectuează în ordine naturală, au loc 300 de înmulțiri.

Să efectuăm același produs în ordinea care rezultă din expresia

$$A_1 \times ((A_2 \times A_3) \times A_4).$$

Efectuând produsul A_2 cu A_3 se efectuează 10 înmulțiri și se obține o matrice cu 1 linie și 1 coloană. Această matrice se înmulțește cu A_4 , se fac 10 înmulțiri și se obține o matrice cu 1 linie și 10 coloane. Dacă o înmulțim pe aceasta cu prima, efectuăm 100 de înmulțiri, obținând rezultatul final cu numai 120 de înmulțiri.

În concluzie, apare o problemă foarte interesantă și anume de a afla *modul în care trebuie să se înmulțească cele n matrice, astfel încât numărul de înmulțiri să fie minim*.

Să vedem, mai întâi, în câte moduri se poate calcula un produs de n astfel de matrice. Pentru $n=4$, putem avea:

$$\begin{aligned} & ((A_1 \times A_2) \times (A_3 \times A_4)); \quad (((A_1 \times A_2) \times A_3) \times A_4); \quad ((A_1 \times (A_2 \times A_3)) \times A_4); \\ & (A_1 \times (A_2 \times (A_3 \times A_4))); \quad (A_1 \times ((A_2 \times A_3) \times A_4)). \end{aligned}$$

Astfel, pentru $n=4$ avem 5 posibilități de calcul al acestui produs. Să observăm că pentru produse de n matrice, sunt necesare $n-1$ perechi de paranteze.



Exercițiu. Puteți demonstra prin inducție acest rezultat?

Tinând cont de aceasta, se poate formula problema următoare: în câte feluri se pot combina n perechi de paranteze (desigur, pentru problema dată avem $n-1$ perechi)? Cei pasionați de informatică pot studia numărarea arborilor binari, autoinstruire (vezi *Capitolul 9*, problema propusă 23). Acum ne limităm să spunem că acest număr este:

$$\frac{1}{n+1} C_{2n}^n$$

și că, în general, un astfel de număr este foarte mare.



Exercițiu. Calculați acest număr pentru $n=10, 11, \dots, 20$.

Prin urmare, un algoritm în care calculăm în toate modurile posibile acest produs este ineficient.

Din fericire, pentru această problemă există o rezolvare polinomială, prin utilizarea programării dinamice. Să presupunem că produsul $A_i \times A_{i+1} \times \dots \times A_j$ s-a calculat optim. În final, s-au înmulțit două matrice $(A_i \times \dots \times A_k) \times (A_{k+1} \times \dots \times A_j)$. Atunci, produsele $A_i \times \dots \times A_k$ și $A_{k+1} \times \dots \times A_j$ au fost calculate optim. De ce? Demonstrați prin reducere la absurd.

Pentru rezolvare, vom aplica principiul al 3-lea al programării dinamice.

În vederea rezolvării problemei, reținem o matrice A cu n linii și n coloane. Elementul $A(i, j)$, $i < j$, reprezintă numărul minim de înmulțiri pentru efectuarea produsului $A_i \times A_{i+1} \times \dots \times A_j$. De asemenea, numărul liniilor și al coloanelor celor n matrice sunt reținute într-un vector DIM cu $n+1$ componente. Pentru exemplul nostru DIM reține următoarele valori: 10, 1, 10, 1, 10.

Pentru rezolvare se ține cont de următoarele relații existente între componente matricei A :

- 1) $A(i, i) = 0$;
- 2) $A(i, i+1) = \text{DIM}(i) \times \text{DIM}(i+1) \times \text{DIM}(i+2)$;
- 3) $A(i, j) = \min_{i \leq k < j} \{A(i, k) + A(k+1, j) + \text{DIM}(i) \times \text{DIM}(k+1) \times \text{DIM}(j+1)\}$

Justificarea acestor relații este următoarea:

- 1) o matrice nu se înmulțește cu ea însăși, deci se efectuează 0 înmulțiri;
- 2) liniile și coloanele matricei A_i se găsesc în vectorul DIM pe pozițiile i și $i+1$, iar ale matricei A_{i+1} , pe pozițiile $i+1$ și $i+2$;
- 3)
 - înmulțind matricele $A_i \times A_{i+1} \times A_k$, se obține o matrice cu un număr de linii egal cu acela al matricei A_i ($\text{DIM}(i)$) și cu un număr de coloane egal cu acela al matricei A_k ($\text{DIM}(k+1)$);
 - înmulțind matricele $A_{k+1} \times \dots \times A_j$, se obține o matrice cu un număr de linii egal cu acela al matricei A_{k+1} ($\text{DIM}(k+1)$) și cu un număr de coloane egal cu acela al matricei A_j ($\text{DIM}(j+1)$);
 - prin înmulțirea celor două matrice se obține matricea rezultat al înmulțirii $A_i \times \dots \times A_j$, iar pentru această înmulțire de matrice se efectuează $\text{DIM}(i) \times \text{DIM}(k+1) \times \text{DIM}(j+1)$ înmulțiri.



Observații

- ✓ Relația sintetizează faptul că pentru a obține numărul de înmulțiri optim pentru produsul $A_i \times \dots \times A_j$ se înmulțesc două matrice, una obținută ca produs optim între $A_i \times \dots \times A_k$ și cealaltă obținută ca produs optim între $A_{k+1} \times \dots \times A_j$, în ipoteza în care cunoaștem numărul de înmulțiri necesare efectuării acestor două produse, oricare ar fi k cuprins între limitele date.
- ✓ Această observație este o consecință directă a programării dinamice și anume că produsul efectuat optim între matricele prezentate se reduce în ultimă instanță la a efectua un produs între două matrice cu condiția ca acestea să fie calculate optim (produsul lor să aibă un număr minim de înmulțiri).
- ➔ Să observăm faptul că orice secvență pentru calculul costului optim este de forma $A_i \times A_{i+1} \times \dots \times A_j$. Astfel, întotdeauna $j \geq i$. Pentru a reține costul optim, $A(i, j)$ vom utiliza o matrice. Cum $j \geq i$ înseamnă că din această matrice se va utiliza numai partea situată deasupra diagonalei principale.
- ➔ Din relația 1, rezultă că mai întâi trebuie completate cu 0, elementele aflate pe diagonala principală. Dacă matricea este declarată ca variabilă globală, elementele ei sunt oricum inițializate cu 0.
- ➔ Din relația 2, rezultă că, în continuare, trebuie completate elementele de coordonate $(i, i+1)$. Toate acestea sunt situate în partea aflată deasupra diagonalei principale, pe o paralelă la diagonala principală ("cea mai apropiată").

- Din relația 3, rezultă că pentru a afla costul optim pentru produsul $A_i \times A_{i+1} \times \dots \times A_j$, reținut de $A(i,j)$, se fac $j-i$ comparații, prin care produsul se descompune în alte două produse, al căror cost optim este deja cunoscut și se alege descompunerea care asigură costul minim. Elementele necesare din matricea costurilor optime se observă în tabelul de mai jos:

k	Produsele	Elemente necesare
i	$(A_1) \times A_{i+1} \times \dots \times A_j$	$A(i,i)$, $A(i+1,j)$
i+1	$(A_i \times A_{i+1}) \times A_{i+2} \times \dots \times A_j$	$A(i,i+1)$, $A(i+2,j)$
...
j-1	$(A_i \times A_{i+1} \times \dots \times A_{j-1}) \times A_j$	$A(i,j-1)$, $A(j,j)$

Din tabel se observă că pentru calculul lui $A(i,j)$ sunt necesare:

- $A(i,i), A(i,i+1), \dots, A(i,j-1)$ - adică toate elementele din matricea $A(i,j)$, aflate pe linia i, până la coloana j.
- $A(i+1,j), A(i+2,j), \dots, A(j,j)$ - adică toate elementele din matricea $A(i,j)$, aflate pe coloana j, până la linia i.

De exemplu, dacă $n=5$, alăturat puteți observa elementele implicate în calculul costului optim $a(2,5)$. **Întrebarea** este: pentru ce au fost prezentate toate aceste amănunte? **Răspuns:** pentru a calcula costul optim, reținut de $A(i,j)$, este necesar ca elementele matricei A să fie completate pe diagonala principală, apoi pe cea mai apropiată paralelă de aceasta, apoi, din nou, pe cea mai apropiată diagonală paralelă cu ultima completată, până se ajunge să se completeze $A(1,n)$ care va reține costul optim pentru problema cerută. În acest fel, *pentru orice element care se calculează, elementele necesare au fost deja determinate*.

Se pune problema să aflăm cum putem efectua acest calcul utilizând relațiile prezentate. Pentru exemplificare vom utiliza exemplul dat la începutul acestui paragraf. Datorită relației 1, diagonala principală a matricei A (cu 4 linii și 4 coloane) va fi alcătuită numai din elemente având valoarea 0.

- Rămâne să determinăm modul de generare a elementelor de pe paralelele la diagonala principală, mai precis pe cele aflate deasupra diagonalei principale. Astfel:

- pentru prima "paralelă": $A(1,2), A(2,3), \dots, A(n-1,n)$;
- pentru a doua "paralelă": $A(1,3), A(2,4), \dots, A(n-2,n)$;
- pentru a treia "paralelă": $A(1,4), A(2,5), \dots, A(n-3,n)$;
- ...
- pentru ultima "paralelă": $A(1,n)$.

Observăm că:

- pentru prima paralelă, liniile sunt între 1 și $n-1$;
- pentru a doua paralelă, liniile sunt între 1 și $n-2$;
- pentru a treia paralelă, liniile sunt între 1 și $n-3$;
- ...
- pentru ultima paralelă liniile sunt între 1 și 1.

Dacă notăm linia cu i și coloana cu j, atunci secvența:

pentru 1 de la 1 la $n-1$
pentru i de la 1 la $n-1$
 $j \leftarrow 1+i$

generează pentru $1=1$ linii de la 1 la $n-1$, pentru $1=2$, linii de la 1 la $n-2$, ..., pentru $1=n-1$, linia 1. Să observăm faptul că pentru fiecare 1 și i, coloana este j. În acest fel, pentru o anumită valoare a lui 1, se obține o paralelă la diagonala principală. De exemplu, dacă $1=1$, $i=1$, $j=2$, se obține $A(1,2)$, dacă $1=2$, $j=3$, se obține $A(2,3)$, ..., iar dacă $1=1$, $i=n-1$, $j=n$, se obține $A(n-1,n)$, adică prima paralelă la diagonala principală.

Revenim la exemplul dat la începutul acestui paragraf.

Inițial se pot calcula numai elementele $A(i,i+1)$, adică $A(1,2), A(2,3), A(3,4)$ - elemente situate pe o paralelă la diagonala principală a matricei A . În concluzie, avem $A(1,2)=100$, $A(2,3)=10$, $A(3,4)=100$. Matricea A va arăta ca alăturat:

$$A = \begin{pmatrix} 0 & 100 & x & x \\ x & 0 & 10 & x \\ x & x & 0 & 100 \\ x & x & x & 0 \end{pmatrix}$$

În continuare calculăm:

$$\begin{aligned} A(1,3) &= \min_{1 \leq k \leq 3} \{A(1,k) + A(k+1,3) + \text{DIM}(1) \times \text{DIM}(k+1) \times \text{DIM}(4)\} = \\ &= \min\{0 + 10 + 10 \times 1 \times 1, 100 + 0 + 10 \times 10 \times 1\} = 20; \\ A(2,4) &= \min_{2 \leq k \leq 4} \{A(2,k) + A(k+1,4) + \text{DIM}(2) \times \text{DIM}(k+1) \times \text{DIM}(5)\} = \\ &= \min\{0 + 100 + 1 \times 10 \times 10, 10 + 0 + 1 \times 1 \times 10\} = 20; \\ A(1,4) &= \min_{1 \leq k \leq 4} \{A(1,k) + A(k+1,4) + \text{DIM}(1) \times \text{DIM}(k+1) \times \text{DIM}(5)\} = \\ &= \min\{0 + 20 + 10 \times 1 \times 10, 100 + 100 + 10 \times 10 \times 10, 20 + 0 + 10 \times 1 \times 10\} = 120; \end{aligned}$$

În concluzie, pentru exemplul nostru, se fac minimum 120 de înmulțiri, rezultat luat din matricea A și anume $A(1,4)$:

$$A = \begin{pmatrix} 0 & 100 & 20 & 120 \\ x & 0 & 10 & 20 \\ x & x & 0 & 100 \\ x & x & x & 0 \end{pmatrix}$$

Mai avem de lămurit o problemă. Chiar dacă cunoaștem costul optim (numărul minim de înmulțiri), cum determinăm în ce mod se înmulțesc matricele pentru a obține acest cost? Să observăm că elementele din matrice, situate sub diagonala principală, au fost neutilizate. De asemenea, pentru calculul optim al unui produs $A_i \times A_{i+1} \times A_{i+2} \times \dots \times A_j$, se determină un anumit k . Așa cum am arătat, k exprimă ordinea de înmulțire a matricelor. De exemplu, dacă $k=i+1$, avem produsul $(A_i \times A_{i+1}) \times A_{i+2} \times \dots \times A_j$, înțelegând prin aceasta că produsul se obține ca produs între matricele $A_i \times A_{i+1}$ și $A_{i+2} \times \dots \times A_j$, produse pe care știm să le obținem în mod optim. Dar, dacă în urma acestui calcul am determinat k , astfel încât produsul obținut să fie optim, această valoare poate fi reținută de $A(j, i)$. Pentru exemplul nostru, matricea A este prezentată mai jos:

$$A = \begin{pmatrix} 0 & 100 & 20 & 120 \\ 1 & 0 & 10 & 20 \\ 1 & 2 & 0 & 100 \\ 1 & 3 & 3 & 0 \end{pmatrix}$$

Grafic, valorile se pot prezenta într-un arbore. Întrucât arborii vor fi descriși într-un capitol separat, vă rog să reveniți după parcurgerea aceluia capitol la această problemă.

Priviți arborele următor:

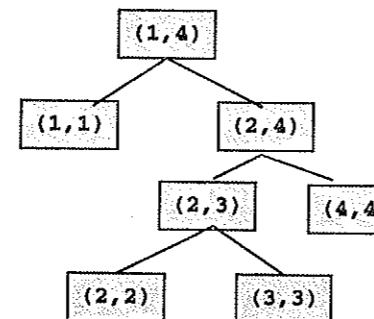


Figura 6.1. Reprezentarea valorilor sub forma unui arbore

Pentru produsul matricelor de la 1 la 4, se obține $k=1$ ($A(4, 1)=1$). Aceasta înseamnă că produsul se va efectua astfel: $A_1 \times (A_2 \times A_3 \times A_4)$. Pentru produsul $A_2 \times A_3 \times A_4$ avem $k=3$. Aceasta înseamnă că produsul va fi calculat astfel: $(A_2 \times A_3) \times A_4$. Prin urmare, produsul celor 4 matrice se va calcula în felul următor:

$$(A_1 \times ((A_2 \times A_3) \times A_4)).$$

Dacă listăm nodurile neterminale ale acestui arbore, arboarele fiind parcurs în postordine, obținem modul în care se așeză parantezele pentru calculul produsului de matrice. Pentru exemplul nostru, subprogramul **parc** afișează $(2, 3)$, $(2, 4)$, $(1, 4)$. Aceasta ne spune cum să aranjăm parantezele.

Varianta Pascal	Varianta C++
<pre> var i,n:longint; dim:array[1..10] of longint; a:array[1..10,1..10] of longint; procedure parc(l,c:longint); var k:integer; begin k:=a[c,l]; if k<>l then parc(l,k); if (k+1)<=c then parc(k+1,c); writeln(l,' ',c); end; procedure costopt; var k,i,j,l,m:longint; begin for l:=1 to n-1 do for i:=1 to n-1 do begin j:=i+1; a[i,j]:=100000; for k:=i to j-1 do begin m:=a[i,k]+a[k+1,j]+ dim[i]*dim[k+1]*dim[j+1]; if a[i,j]>m then begin a[i,j]:=m; a[j,i]:=k; end end end; writeln ('cost optim ',a[1,n]); end; end; begin write('n='); readln(n); for i:=1 to n+1 do begin write('d='); readln(dim[i]); end; costopt; parc(1,n); end. </pre>	<pre> #include <iostream.h> long i,n,dim[10],a[10][10]; void parc (int l, int c) { int k=a[c][l]; if (k!=l) parc(l,k); if (k+1==c) parc(k+1,c); cout<<l<< " "<<c<<endl; } void costopt () { long k,i,j,l,m; for (l=1;l<=n-1;l++) for (i=1;i<=n-1;i++) { j=i+1; a[i][j]=100000; for (k=i;k<=j-1;k++) { m=a[i][k]+a[k+1][j]+ dim[i]*dim[k+1]*dim[j+1]; if (a[i][j]>m) { a[i][j]=m; a[j][i]=k; } } cout<<"cost optim " <<a[1][n]<<endl; } main() { cout<<"n="; cin>>n; for (i=1;i<=n+1;i++) { cout<<"d="; cin>>dim[i]; } costopt(); parc(1,n); } </pre>

Complexitatea algoritmului este $O(n^3)$. Observați că există trei cicluri **for** imbricate!

6.8. Probleme cu ordinea lexicografică a permutărilor

Problema 1. Se citește n , număr natural, și o permutare a numerelor $1, 2, \dots, n$. Se cere să se afișeze numărul de ordine al permutării, dacă se consideră permutările în ordine lexicografică. Alăturat, observați permutările mulțimii $\{1, 2, 3\}$, listate în ordinea lexicografică.

N=3
1 123
2 132
3 213
4 231
5 312
6 321

Prima idee este să generăm prin **Backtracking**, în ordine lexicografică, toate permutările, până la întâlnirea permutării date și în paralel să le numărăm. Când am întâlnit-o, afișăm numărul de ordine. Dar, avem $n!$ permutări. Cum $n!=1\times 2\times \dots \times n > 2\times 2\times 2\dots 2 = 2^{n-1}$, observăm că algoritmul este exponențial. Evident, renunțăm la o astfel de soluție.

Rezolvare. Să observăm primul element afișat al fiecărei permutări. Astfel, există două permutări care afișează 1 ca prim element, două permutări care afișează 2, ca prim element și 2 permutări care afișează 3 ca prim element. Dacă analizăm și sirul permutărilor observăm că, în general: elementul 1 este afișat de $(n-1)!$ ori, elementul 2 este afișat de $(n-1)!$ ori, ..., elementul n este afișat de $(n-1)!$ ori.

De fapt,

$$\underbrace{(n-1)! + (n-1)! + \dots + (n-1)!}_{de\ n\ ori} = n!.$$

Această simplă observație ne permite să scriem o relație de recurență, unde $NR(P(n))$ înseamnă numărul de ordine al permutării inițiale, iar $NR(P(n-1))$ înseamnă numărul de ordine al permutării alcătuite din ultimele $n-1$ elemente ale permutării inițiale. Dacă permutarea este a_1, a_2, \dots, a_n , atunci:

$$NR(P(n)) = \begin{cases} (n-1)!(a_1 - 1) + NR(P(n-1)), & n > 1 \\ 1 & n = 1 \end{cases}$$

Să observăm faptul că permutarea formată cu ultimele $n-1$ elemente ale permutării inițiale este formată din elementele de la 1 la n , din care lipsește exact un element. Din ea, se poate obține permutarea cu același număr de ordine (ordinea lexicografică) a mulțimii $\{1, 2, \dots, n-1\}$, dacă din orice element mai mare decât cifra eliminată se scade 1 (această operație nu afectează ordinea lexicografică).

Exemplu: pentru $n=5$, $P=4\ 5\ 2\ 3\ 1$.

$$P=4!3+NR(\{5, 2, 3, 1\})=72+NR(\{4, 2, 3, 1\})=72+3!*3+NR(\{2, 3, 1\})=90+2!1+NR(\{3, 1\})=92+NR(\{2, 1\})=92+1!1+P(1)=93+1=94.$$

Varianța Pascal	Varianța C++
<pre>var P:array[1..10] of integer; n,i,k,Nr,Prod:integer; procedure Numar; var nsalv:integer; begin nsalv:=n; while n>1 do begin Prod:=Prod div n; Nr:=Nr+(P[k]-1)*Prod; for i:=k+1 to nsalv do if P[i]>P[k] then P[i]:=P[i]-1; k:=k+1; n:=n-1; end; Nr:=Nr+1; end; begin write ('n='); readln(n); for i:=1 to n do readln(P[i]); Prod:=1; for i:=2 to n do Prod:=Prod*i; k:=1; Numar; writeln(Nr); end.</pre>	<pre>#include <iostream.h> int P[10],n,i,k,Nr,Prod; void Numar() { int nsalv=n; while (n>1) { Prod/=n; Nr+=(P[k]-1)*Prod; for (i=k+1;i<=nsalv;i++) if (P[i]>P[k]) P[i]--; k++;n--; } Nr++; } main() { cout<<"n=">>n; for(i=1;i<=n;i++) cin>>P[i]; Prod=1; for(i=2;i<=n;i++) Prod*=i; k=1; Numar(); cout<<Nr; }</pre>

Algoritmul are complexitatea $O(n^2)$. Justificați!

Problema 2. Se citește $n > 0$, număr natural, și un număr natural $1 \leq Nr \leq n!$. Care este permutarea care, în ordine lexicografică, are numărul de ordine Nr ?

Exemplu: pentru $n=5$ și $Nr=94$, se va afișa $4\ 5\ 2\ 3\ 1$.

Rezolvare. Știm că, în sirul tuturor permutărilor, fiecare element al permutării apare pe prima poziție de exact $(n-1)!$ ori. În cazul nostru, $(n-1)!=4!=24$. Cum primele (în ordine lexicografică) $(n-1)!$ permutări au elementul 1 pe prima poziție, următoarele $(n-1)!$ permutări au elementul 2 pe prima poziție, înseamnă că:

$$\text{Primul_element}=1+[(Nr-1)/(n-1)!],$$

unde prin $[x]$ am notat parte întreagă din x .

Observație foarte importantă. Dacă primul element al permutării poate fi depistat ca mai sus, cu următoarele trebuie efectuate operații în plus, pentru că se selectează elementele unei permutări ale cărei elemente nu sunt numere consecutive. De exemplu, dacă pentru $n=5$ se selectează ca prim element numărul 3, atunci următorul element trebuie selectat din mulțimea $\{1, 2, 4, 5\}$. În acest caz, algoritmul va furniza indicele elementului în vector și nu elementul propriu-zis. Astfel, dacă elementele ar ocupa în vectorul P poziții consecutive, acesta ar trebui să fie $P(1, 2, 4, 5)$ și dacă algoritmul returnează 3, ar trebui selectat 4.

Datorită celor arătate, vom prefera să lucrăm cu 2 vectori, ca alăturat. Vectorul P reține numerele 1, 2, ..., n , iar $O[i]$ reține 1 dacă elementul corespunzător din P a fost selectat, și 0 în caz contrar. Astfel, dacă algoritmul returnează valoarea k , va trebui afișat al k -lea element neselectat, după care $O[k]$ va reține 1. Exemplul care urmează vă va lămuri.

Ex: Avem $n=5$ și $Nr=94$.

$(94-1)/24+1=3+1=4$. Numărul rămas este $94-3*24=94-72=22$. Pentru că elementul selectat este 4 (al 4-lea în ordine), marcăm în vectorul O acest fapt. În acest mod, problema s-a redus la o alta, mai simplă: care este permutarea cu 4 elemente, 1, 2, 3, 5 care, în ordine lexicografică, are numărul de ordine 22?

$P \rightarrow$	1	2	3	4	5
$O \rightarrow$	0	0	0	1	0

Avem: $(n-1)!=3!=6$. La fel: $(22-1)/6+1=3+1=4$. Al 4-lea element neselectat este 5. Îl selectăm pentru a-l afișa și-l marcăm în O . Până în acest moment am afișat 4 și 5. Numărul rămas este $22-3*6=22-18=4$.

$P \rightarrow$	1	2	3	4	5
$O \rightarrow$	0	0	0	1	1

Problema s-a redus la o alta, mai simplă: care este permutarea cu 3 elemente, 1, 2, 3 care, în ordine lexicografică, are numărul de ordine 4?

$$(n-1)! = 2! = 2$$

$$(4-1)/2+1 = 2$$

Numărul rămas este $4-1*2=2$.

Selectăm al doilea element neselectat, adică 2 și afișăm elementul. Astfel, am afișat 4, 5, 2.

$P \rightarrow$	1	2	3	4	5
$O \rightarrow$	0	1	0	1	1

Apoi:

$$(n-1)! = 1! = 1$$

$$(2-1)/1+1 = 2$$

$P \rightarrow$	1	2	3	4	5
$O \rightarrow$	0	1	1	1	1

Numărul rămas este $2-1*1=1$. Afișăm 3, pentru că este al doilea element neselectat, după care marcăm în O elementul 3. Am afișat astfel 4, 5, 2 și 3.

Afișăm ultimul element neafiat, adică 1. Am obținut 4, 5, 2, 3 și 1.

Varianta Pascal

```
var n,Nr,i,k,fact,ic,
    nsalv:integer;
    P,O:array[1..10] of
        integer;
begin
    write ('n='); readln(n);
    write ('Nr='); readln(Nr);
    fact:=1;
    for i:=1 to n do
    begin
        P[i]:=i;
        fact:=fact*i;
    end;
    nsalv:=n;
    while Nr>>1 do
    begin
        fact:=fact div n;
        n:=n-1;
        k:=(Nr-1) div fact+1;
        Nr:=Nr-fact*(k-1);
        ic:=0;i:=1;
        while ic<k do
        begin
            if O[i]=0 then
                ic:=ic+1;
            i:=i+1;
        end;
        O[i-1]:=1;
        write(P[i-1]);
    end;
    for i:=1 to nsalv do
    if O[i]=0 then
        write (P[i]);
end.
```

Varianta C++

```
#include <iostream.h>
int n,Nr,i,k,fact,ic,
    nsalv,P[10],O[10];
main()
{ cout<<"n="; cin>>n;
cout<<"Nr="; cin>>Nr;
fact=1;
for(i=1;i<=n;i++)
{ P[i]=i;
    fact*=i;
}
nsalv=n;
while (Nr!=1)
{ fact/=n;
    n--;
    k=(Nr-1)/fact+1;
    Nr-=fact*(k-1);
    ic=0;i=1;
    while(ic<k)
    { if (O[i]==0) ic++;
        i++;
    }
    O[i-1]=1;
    cout<<P[i-1];
}
for(i=1;i<=nsalv;i++)
if(O[i]==0) cout<<P[i];
}
```

Algoritmul are complexitatea $O(n^2)$. Justificați!

6.9. Numărul partițiilor unei mulțimi cu n elemente

Enunț. Se citește n , număr natural. Se cere ca programul să afișeze numărul partițiilor unei mulțimi cu n elemente.

Rezolvare. Prima idee care ne vine în minte este să generăm toate partițiile unei mulțimi (vedeți generarea lor prin **Backtracking**) și să le numărăm. Dar numărul partițiilor este aşa de mare astfel încât o asemenea idee se dovedește dezastruasă. Pornind de la partițiile unei mulțimi cu n elemente, observați, pentru $n=3$, cum se pot obține toate partițiile unei mulțimi cu $n+1$ elemente.

{1, 2, 3}	(1, 2, 3), {4} (1, 2, 3, 4)
{1} {2, 3}	{1, 4}, {2, 3} (1), {2, 3, 4} (1), {2, 3}, {4}
{2} {1, 3}	{2, 4}, {1, 3} (2), {1, 3, 4} (2), {1, 3}, {4}
{3} {1, 2}	{3, 4}, {1, 2} (3), {1, 2, 4} (3), {1, 2}, {4}
{1} {2} {3}	{1, 4}, {2}, {3} (1), {2, 4}, {3} (1), {2}, {3, 4} (1), {2}, {3}, {4}

Ideea de bază este ca, din fiecare partitie a mulțimii de n elemente, să se genereze toate partitiile care provin din ea, ale mulțimii de $n+1$ elemente. Aceasta se obține dacă se adaugă pe rând, la fiecare mulțime a partitiei, elementul $n+1$ și, la sfârșit, acesta va forma singur o mulțime a partitiei. Vedeti mai sus!

Vom nota prin $S(n, k)$ numărul partitiilor unei mulțimi cu n elemente, partitii în care numărul mulțimilor este k . De exemplu, pentru $n=3$, avem: $S(3, 1)=1$, $S(3, 2)=3$ și $S(3, 3)=1$. În aceste condiții, numărul partitiilor unei mulțimi cu 3 elemente este: $S(3, 1)+S(3, 2)+S(3, 3)=1+3+1=5$. De aici rezultă că, pentru a calcula numărul partitiilor unei mulțimi cu $n+1$ elemente, trebuie să calculăm suma:

$$S(n+1, 1)+S(n+1, 2)+\dots+S(n+1, n+1).$$

Dacă am găsi o modalitate de calcul pentru $S(n, k)$, atunci problema ar fi rezolvată.

a) Să observăm că, pentru orice n , $S(n, 1)=1$, adică avem o singură partitie în care toate mulțimile au un singur element: {1}, {2}, {3}, ..., {n}. Tot așa, $S(n, n)=1$, pentru că avem o singură partitie în care mulțimile au n elemente: {1, 2, ..., n}.

b) Urmărim să găsim o relație de recurență pentru $S(n+1, k)$, adică să găsim numărul partitiilor unei mulțimi cu $n+1$ elemente, partitii în care toate mulțimile au k elemente, în condițiile în care cunoaștem $S(n, 1)$, $S(n, 2)$..., $S(n, n)$.

b1) Dacă cunoaștem numărul partitiilor unei mulțimi cu n elemente, în care fiecare partitie are k submulțimi, $S(n, k)$, atunci putem forma partitii alcătuite din exact k mulțimi ale mulțimii cu $n+1$ elemente. Pentru fiecare partitie numărată de $S(n, k)$, adăugăm elementul $n+1$ în prima mulțime a partitiei, apoi în a doua mulțime, ..., la sfârșit în a k -a mulțime a partitiei. În acest fel, din fiecare partitie se obțin alte k partitii. În concluzie, vom obține în acest mod $k \cdot S(n, k)$ partitii. În exemplul dat, observați cum din cele 3 ($S(3, 2)$) partitii cu 2 submulțimi ale mulțimii cu 3 elemente, am obținut $2 \cdot S(3, 2)=6$ partitii cu 3 submulțimi ale unei mulțimi cu 4 elemente.

{1} {2, 3}	(1, 4), {2, 3} (1), {2, 3, 4}
{2} {1, 3}	{2, 4}, {1, 3} (2), {1, 3, 4}
{3} {1, 2}	{3, 4}, {1, 2} (3), {1, 2, 4}

b2) Dacă cunoaștem numărul partitiilor unei mulțimi cu n elemente, partitii care sunt alcătuite din $k-1$ mulțimi, atunci din fiecare astfel de mulțime se poate obține o altă partitie cu k mulțimi ale mulțimii cu $n+1$ elemente, adăugând la fiecare partitie mulțimea alcătuită din elementul $n+1$. În concluzie, în acest mod vom obține alte $S(n, k-1)$ partitii cu k submulțimi ale unei mulțimi cu $n+1$ elemente. În exemplul dat, avem:

{1, 2, 3} {1, 2, 3} {4}

Cum alte posibilități de obținere a partitiilor unei mulțimi cu k clase ale unei mulțimi cu $n+1$ elemente nu există și cum, astfel obținute, partitiile nu se repetă, relația este:

$$S(n+1, k)=S(n-1, k)+k \cdot S(n, k).$$

Pentru exemplul dat, $S(4, 2)=S(3, 3)+3 \cdot S(3, 2)=1+2 \cdot 3=7$.

Pentru a scrie programul, completăm, mai întâi prima coloană a matricei cu 1 (pentru că avem $S(n, 1)=1$). Prima linie va avea numai un 1 în prima coloană, pentru că, evident, $S(1, k)=0$, pentru $k>1$. În continuare, completăm pe linii elementele matricei S . În final, facem suma pe linia n .

Varianța Pascal	Varianța C++
<pre>var S:array[1..20,1..20] of longint; n,k,i,j,suma:longint; begin write('n='); readln(n); for i:=1 to n do S[i,1]:=1; for i:=2 to n do for j:=2 to i do S[i,j]:=S[i-1,j-1]+j*S[i-1,j]; for i:=1 to n do suma:=suma+S[n,i]; writeln(suma) end.</pre>	<pre>#include <iostream.h> long S[20][20],n,k,i,j,s; main() { cout<<"n="; cin>>n; for (i=1;i<=n;i++) S[i][1]=1; S[i][1]=1; for(i=2;i<=n;i++) for (j=2;j<=i;j++) S[i][j]=S[i-1][j-1]+j*S[i-1][j]; for (i=1;i<=n;i++) s+=S[n][i]; cout<<s; }</pre>

Complexitatea algoritmului este $O(n^2)$.

Exercițiu. Modificați programul de așa natură astfel încât să afișeze primele n linii ale tabelului $S(n, k)$. Nu afișați prea multe linii, pentru că numerele devin foarte mari.

Probleme propuse

1. Dintr-un element $(a_{i,j})$ al unei matrice $A_{n,n}$ se poate ajunge în elementele $a_{i+1,j}, a_{i+1,j+1}, a_{i+1,j-1}$. Știind că fiecare element al matricei reține un număr natural, se cere un drum care îndeplinește condițiile problemei și unește un element de pe linia 1 cu unul de pe linia n astfel încât suma numerelor reținute de elementele pe unde trece drumul să fie maximă.

Exemplu: Pentru $n=3$ și matricea $A = \begin{pmatrix} 3 & 1 & 3 \\ 4 & 1 & 2 \\ 3 & 7 & 1 \end{pmatrix}$,

drumul este: $a_{1,1}, a_{2,1}, a_{3,2}$, iar suma este $3+4+7=14$.

Rezolvați problema prin utilizarea a 3 metode:

- Greedy heuristic (euristică)** - se obține o soluție "suficient de bună", deși de cele mai multe ori, neoptimă. Se alege un element de maxim de pe linia 1, apoi, pornind de la el, cel mai mare element accesibil de pe linia 2, ... Care este complexitatea algoritmului în acest caz?
- Backtracking;**
- Prin **programare dinamică** - în acest caz, să se rezolve problema prin **metoda înainte** și prin **metoda înapoi**.

2. **Problema diligenței.** Un comis voiajor are de făcut o călătorie cu diligența între două orașe din vestul sălbatic. Călătoria cu diligența se desfășoară în n etape. În fiecare etapă diligența merge, fără întrerupere, între două orașe. După o etapă, diligența este schimbată, iar comis-voiajorul decide în care oraș va merge în etapa următoare. Se știe că în fiecare etapă, cu excepția ultimei etape, se poate ajunge în unul dintre cele k orașe. Se cunoaște orașul de start, s și cel final, F (în ultima etapă se ajunge în orașul F). Pentru orice etapă se cunosc toate distanțele între orașele care sunt puncte de pornire și cele care sunt puncte de destinație.

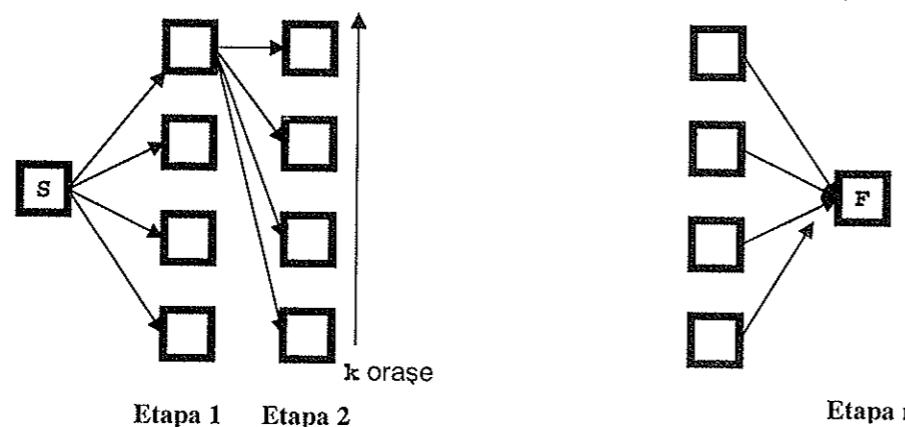


Figura. 6.2. Schemă de principiu

Se cere să se decidă care sunt orașele destinație pentru o anumită etapă, astfel încât lungimea totală a drumului, care trebuie afișată, să fie minimă.

Marius Popescu

3. **"Lucrare de control".** Mai multor elevi li se cere să pună într-o anumită ordine un număr de $n < 200$ cuvinte formate numai din litere mici - ordinea exprimă faptul că aceste cuvinte se succed după o anumită logică.

Exemplu: platon kant marx stalin havel.

Logica succesiunii constă în faptul că fiecare cuvânt poate fi definit folosind numai cuvintele anterioare.

Fiecare elev îi transmite profesorului succesiunea de cuvinte care i se pare logică. Sarcina profesorului constă în a acorda fiecărui elev una dintre notele 1, 2, 3, ..., n , corespunzătoare numărului de cuvinte așezate într-o succesiune corectă. Pentru exemplul 1 avem următoarele note:

marx stalin kant platon havel	3
havel marx stalin kant platon	2
havel stalin marx kant platon	1

Horia Georgescu, O.N.I.

4. Un patron de gogoșerie a cumpărat un calculator și dorește să învețe să lucreze pe el. Pentru aceasta va umple un raft de cărți dintr-o anumită serie. Raftul are lungimea L cm (L este număr natural). Seria dispune de n titluri 1, 2, ..., n cu grosimiile de n_1, n_2, \dots, n_n cm (numere naturale).

Să se selecteze titlurile pe care le va cumpăra patronul astfel încât raftul să fie umplut complet (suma grosimilor cărților cumpărate să fie egală cu lungimea raftului) și numărul cărților achiziționate să fie maxim.

5. **Algoritm performant, în $O(n \times \log(n))$, pentru subșir crescător de lungime maximală.** Vom prezenta un algoritm, mai performant, pentru determinarea unui subșir crescător de lungime maximală.

Ideea de bază este de a afla, pentru fiecare $v[i]$, numărul de elemente din sir care îl preced și sunt mai mici decât el.

Fie $v=(6, 5, 7, 3, 8, 7, 9)$. Vectorul s reține numere din v prin următoarea regulă: dacă $s[i]$ este $v[k]$, atunci există în v un subșir crescător de lungime i , al cărui ultim element este $v[k]$. Pentru a obține acest rezultat, regula de completare a lui s este de a scrie un număr peste primul element din s care este mai mare sau egal cu el. Dacă nu există un astfel de element, atunci elementul este adăugat lui s la sfârșit. În același timp, după ce un element a fost adăugat în s , indicele lui din s este reținut de vectorul L pe poziția din v a numărului adăugat.

Pentru exemplul nostru:

```
citim 6 S=(6); L=(1);
citim 5 S=(5); L=(1,1);
citim 7 S=(5,7); L=(1,1,2);
citim 3 S=(3,7); L=(1,1,2,1);
citim 8 S=(3,7,8); L=(1,1,2,1,3);
citim 7 S=(3,7,8); L=(1,1,2,1,3,2);
citim 9 S=(3,7,8,9); L=(1,1,2,1,3,2,4).
```

Acum putem reconstitui, în ordine inversă, subșirul crescător de lungime maximă, pornind de la L . Valoarea maximă în L este 4 și are în L indicele 7. $V[7]=9$. Următoarea valoare este 3 și are în L indicele 5. $V[5]=8$. Următoarea valoare este 2 și are în L indicele 3. $V[3]=7$. Următoarea valoare este 1 și are în L indicele 2. $V[2]=5$. Prin urmare, subșirul crescător de lungime maximă este 5, 7, 8, 9.

 Observați că $(3, 7, 8, 9)$ reținut de S nu este subșir crescător, dar numărul de elemente din S este egal cu lungimea unui subșir crescător de lungime maximă. De ce? Fie i o poziție completată a lui s . Prin logica modului de completare a lui s , un element, care există la un moment dat, pe poziția $i-1$, și care se află în v înaintea elementului memorat în s pe poziția i , poate fi înlocuit de alt element care se află în v după elementul memorat în s pe poziția i .

Să analizăm complexitatea algoritmului. Fiecare dintre cele n numere reținute de v este adăugat în vectorul s . Ar trebui parcurs s pentru a adăuga numărul în poziția corespunzătoare. Dar, atenție, în s numerele sunt în ordine crescătoare. Prin urmarea, se poate efectua o căutare binară în $O(\log n)$. Identificarea subșirului se face în $O(n)$. Rezultă complexitatea finală $O(n \times \log n)$.

Cerință. Scrieți un program care implementează algoritmul prezentat. Intrările și ieșirile se vor face prin utilizarea fișierelor text.

6. Mere-pere. Se consideră n camere distincte, situate una după alta, astfel încât din camera i ($i \in \{1, 2, \dots, n-1\}$) se poate trece doar în camera $i+1$. În fiecare cameră se găsesc mere și pere în cantități cunoscute (bucăți).

O persoană cu un rucsac suficient de încăpător, initial gol, pornește din camera 1, trece prin camerele 2, 3, ..., n șiiese. La intrarea în fiecare cameră descarcă rucsacul, și încarcă, fie toate meroile, fie toate perele din cameră respectivă, după care trece în camera următoare. Se presupune că pentru fiecare fruct transportat între două camere, persoana consumă o calorie. Ce fel de fructe trebuie să încarce persoana în rucsac astfel încât după parcurgerea celor n camere să consume un număr minim de calorii și care este acest număr?

Horia Georgescu

7. Se citesc n , număr natural și n numere naturale. Se cere să se afișeze cea mai mare sumă care se poate forma cu numerele dintre cele n (fiecare număr poate participa o singură dată la calculul sumei) și care se divide cu n . Afipați, de asemenea, și numerele care alcătuiesc suma.

8. Se citesc n multimi alcătuite din cel mult 200 de numere naturale între 0 și 199. Se cere să se determine un număr maxim de multimi cu proprietatea că între oricare două există o relație de incluziune. De exemplu, dacă se citesc 4 multimi $\{2, 5, 6\}$, $\{1, 6, 9, 2\}$, $\{2\}$, $\{2, 6\}$ se tipărește 3, adică numărul submultimilor $\{2, 5, 6\}$, $\{2\}$, $\{2, 6\}$.

9. Problema care urmează prezintă o modalitate de compactare a unui fișier oarecare (de date, executabil, etc.). După cum se știe, fișierul este reținut pe suport ca o succesiune de 0 și 1. Noi disponem de un set de m secvențe de 0 și 1 din care nu lipsesc secvența care îl conține numai pe 0 și secvența care îl conține numai pe 1. Se cere să găsim o descompunere a sirului de 0 și 1 care alcătuiește fișierul într-un număr minim de secvențe din cele m . Dacă rezolvăm această problemă, putem înlocui secvența cu o adresă și în acest fel fișierul va deveni o succesiune de adrese (în cazul existenței unor secvențe lungi se realizează economia de suport).

Exemplu: Pentru $m=6$, secvențele sunt:

- 1) 0
- 2) 1
- 3) 0 0 1
- 4) 1 0 0
- 5) 1 1 1
- 6) 1 0 1

Fișierul este: 100111001.

Se folosesc secvențele 4 5 și 3.

10. Pentru un triunghi ABC , cu vârfurile de coordonate întregi, definim costul său ca fiind minimul arilor dreptunghiurilor cu laturile paralele cu axele de coordonate care au pe laturi vârfurile unui triunghi. De exemplu, cu vârfurile de coordonate $(1, 0)$, $(5, 0)$, $(0, 3)$ costul atașat este egal cu 15. Fie un poligon convex, cu coordonatele vârfurilor numere întregi. Numim **triangularizare** a poligonului o partionare a sa ale cărei vârfuri sunt vârfuri ale poligonului dat. Numim costul unei triangularizări ca fiind suma costurilor triunghiurilor componente. Problema constă în realizarea unei triangularizări de cost minim.

Indicații

1. Vezi "problema triunghiului"!
2. Ca la "problema triunghiului". Se calculează drumurile optime corespunzătoare etapei n , apoi $n-1$, ...

3. Subşir crescător de lungime maximă.

4. Vezi "o problemă cu sume"!

6. Aparent, se alege minimul dintre mere și pere din fiecare cameră. Să analizăm exemplul următor:

1	2
M	3 11
P	4 14

Dacă luăm merele din prima cameră, în a doua cameră vom avea 14 mere și 14 pere. Orice am alege, costul este $3+14=17$.

Dar dacă luăm perele, atunci în a doua cameră vom avea 11 mere și 18 pere. Alegem merele și avem costul $4+11=15<17$. În concluzie, rezolvarea prin alegerea minimului nu este corectă.

Dacă ar fi să calculăm minimul după toate variantele de tip **MP...PMPM**, avem un algoritm în $O(2^n)$. Acest lucru se observă ușor dacă, de exemplu, în loc de **M** punem 0 și în loc de **P** punem 1.

Fie M_i , $i=1\dots n$, merele care se află inițial în camera i .

Fie P_i , $i=1\dots n$, perele care se află inițial în camera i .

Vom nota CM_i , $i=1, \dots, n$, costul optim dacă se pleacă cu merele din camera i ($i=1\dots n$).

Vom nota CP_i , $i=1, \dots, n$, costul optim dacă se pleacă cu perele din camera i ($i=1, \dots, n$).

Inițial, avem:

$$CM_n = M_n;$$

$$CP_n = P_n.$$

Pentru $i=1, 2, \dots, n-1$ avem:

$$CM_i = \min \begin{cases} M_i + CP_{i+1} \\ M_i + (M_i + M_{i+1}) + CP_{i+2} \\ M_i + (M_i + M_{i+1}) + (M_i + M_{i+1} + M_{i+2}) + CP_{i+3} \\ \dots \\ M_i + (M_i + M_{i+1}) + (M_i + M_{i+1} + M_{i+2}) + \dots + (M_i + M_{i+1} + \dots + M_n) \end{cases}$$

$$CP_i = \min \begin{cases} P_i + CM_{i+1} \\ P_i + (P_i + P_{i+1}) + CM_{i+2} \\ P_i + (P_i + P_{i+1}) + (P_i + P_{i+1} + P_{i+2}) + CM_{i+3} \\ \dots \\ P_i + (P_i + P_{i+1}) + (P_i + P_{i+1} + P_{i+2}) + \dots + (P_i + P_{i+1} + \dots + P_n) \end{cases}$$

Semnificația relațiilor de recurență. De exemplu pentru CM_1 . Caut minimul de calorii pentru acțiunile:

- se iau merele din camera i , apoi din camera $i+1$ se iau perele;
- se iau merele din camera i , apoi din camera $i+1$ tot merele, iar din camera $i+1$ se iau perele;
- ...
- se iau merele din toate camerele.

Pentru CP_i se procedează în mod identic. În final, se calculează $\min\{CM_1, CP_1\}$ și rezultatul este soluția problemei.

7. Se calculează S , suma tuturor numerelor. Se caută cea mai mare sumă mai mică sau egală cu S , care se divide cu n . Se caută ca la "o problemă cu sume" numerele care însumate o alcătuiesc. Dacă această sumă nu se poate forma, se încearcă următoarea, mai mică decât aceasta care se divide cu n , până când se găsește o asemenea sumă. Se demonstrează faptul (cum?) că există întotdeauna o astfel de sumă.

8. Se sortează lexicografic vectorii caracteristici și apoi se găsește subşirul maximal...

9. Ca la înmulțirea optimală a unui sir de matrice. Notăm cu $Cost[i, j]$ numărul de secvențe necesar codificării textului între caracterele cu numere de ordine i și j . Avem: $Cost[i, i]=1$.

$$Cost[i, j] = \begin{cases} 1, & \text{dacă este o secvență dată} \\ \min_{l \in \{i+1, \dots, j-1\}} \{cost[i, l] + cost[l+1, j]\}, & \text{altfel} \end{cases}$$

Completarea matricei se face pe paralele la diagonala principală:

$cost[1, 1], \dots, cost[n, n], cost[1, 2], \dots, cost[n-1, n], \dots, cost[1, n]$.

Soluția este $cost[1, n]$.

10. Se reține costul minim triangularizării pentru poligoanele formate din 3, 4, ..., $k-1$ puncte, luate în sensul de parcursare a poligonului. Atunci când calculează costul minim al triangularizării pentru poligonul cu k puncte, se alege minimul dintre costul poligonului anterior format, la care se adaugă costul noului triunghi, fie costul obținut prin unirea punctelor 1, 2, ..., $k-1$ cu punctul k .

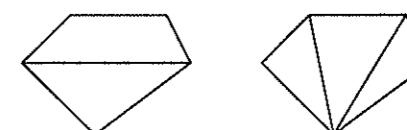


Figura 6.3. Exemple de triangularizare

Capitolul 7

Grafuri neorientate

7.1. Introducere

Uneori, algoritmii trebuie să prelucreze **date referitoare la anumite elemente între care există anumite relații**. Să analizăm exemplele următoare:

1. Se dau n orașe. Unele dintre ele sunt unite prin șosele directe (care nu mai trec prin alt oraș).
2. Se cunosc relațiile de prietenie dintre n persoane.
3. Se dau n țări și se cunoaște relația de vecinătate între ele.
4. Se dau n triunghiuri, iar unele dintre ele sunt asemenea.

Pentru fiecare dintre aceste exemple se poate imagina o reprezentare grafică care să exprime relațiile existente.

→ Convenim ca fiecare element să-l numim **nod** sau **vârf**.

Astfel, în cazul 1. nodul este orașul, în cazul 2. nodul este persoana, în cazul 3. nodul este țara și în cazul 4. nodul este triunghiul. Convenim ca un nod (vârf) să-l notăm cu un cerculeț în care să înscrim numărul lui (de la 1 la n).

→ Relația existentă între două noduri o vom reprezenta grafic unindu-le printr-un segment de dreaptă. Convenim ca un astfel de segment să-l numim **muchie** și dacă ea unește nodurile i și j , să-o notăm cu (i, j) .

În cazul 1., muchia (i, j) are semnificația că între orașele i și j există o șosea directă. În cazul 2., muchia (i, j) are semnificația că persoanele i și j sunt prietene, în cazul 3. muchia (i, j) are semnificația că țările i și j sunt vecine, iar în cazul 4., că triunghiurile i și j sunt asemenea.

Procedând așa, obținem o descriere grafică precum cea din figura 7.1 și convenim ca o astfel de reprezentare să-o numim **graf neorientat**.

Desigur, această abordare este intuitivă. Teoria grafurilor este fundamentată matematic și în cele ce urmează ne propunem să-o prezentăm sistematic.

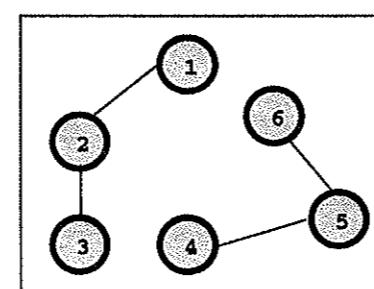


Figura 7.1.
Exemplu de graf neorientat

7.2. Definiția grafului neorientat

Definiția 7.1.¹ Un **graf neorientat** este o pereche ordonată $G = (V, E)$, unde:

→ $V = \{v_1, v_2, \dots, v_n\}$ este o mulțime finită și nevidă. Elementele mulțimii V se numesc **noduri** (vârfuri).

→ E este o mulțime finită de **perechi neordonate** de forma (v_i, v_j) , unde $i \neq j$, și $v_i, v_j \in V$. Elementele mulțimii E se numesc **muchii**. Semnificația unei muchii este aceea că unește două noduri.

Un graf poate fi desenat așa cum se observă în exemplul următor (vezi figura 7.2), unde

$$G = (V, E),$$

$$\begin{aligned} V &= \{1, 2, 3, 4, 5, 6\}; \\ E &= \{(1, 2), (1, 3), (1, 5), (2, 3), (3, 4), \\ &\quad (4, 5)\} \end{aligned}$$

Notăție: În graful $G = (V, E)$, vom nota cu n numărul nodurilor și cu m numărul muchiilor.

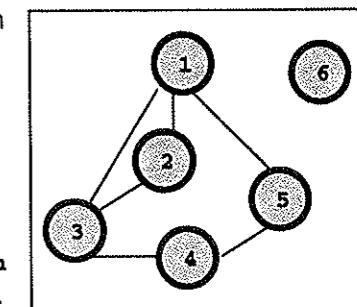


Figura 7.2.
Alt exemplu de graf neorientat

Observații

- ✓ Două noduri distincte pot fi unite prin cel mult o muchie. În exemplul de mai sus, $(1, 2)$ este muchia care unește nodul 1 cu nodul 2. Dacă scriem $(2, 1)$, ne referim la aceeași muchie (**perechea este neordonată**).
- ✓ Nu există o muchie care unește un nod cu el însuși (o muchie unește două noduri distincte).



Definiția 7.2. În graful $G = (V, E)$, nodurile distincte $v_i, v_j \in G$ sunt **adiacente** dacă există muchia $(v_i, v_j) \in E$.

Vom spune că muchia $(v_i, v_j) \in E$ este **incidentă** la nodurile v_i și v_j .

În exemplul dat anterior, nodurile 1 și 5 sunt adiacente, dar nodurile 2 și 5 nu sunt adiacente. Muchia $(4, 5)$ este incidentă la nodurile 4 și 5.



Definiția 7.3. Într-un graf neorientat, prin **gradul** unui nod v se înțelege numărul muchiilor **incidente** cu nodul v și se notează cu $d(v)$. Un nod cu gradul 0 se numește **nod izolat**, iar unul cu gradul 1 se numește **nod terminal**.

În exemplul dat, $d(2)=2$, $d(1)=3$, $d(6)=0$ (6 este nod izolat).

¹ Definiția este restrictivă, în unele lucrări veți întâlni definiții mai puțin restrictive, de exemplu, poate exista o muchie de la un nod la el însuși sau nu se cere ca mulțimea nodurilor să fie finită.

O relație utilă: fie un graf neorientat cu n noduri și m muchii. Dacă notăm cu d_1, d_2, \dots, d_n gradele celor n noduri, atunci avem relația:

$$d_1 + d_2 + d_3 + \dots + d_n = 2m.$$

→ **Demonstrație:** fiecare muchie face să crească gradele celor două noduri la care este incidentă cu câte o unitate. Prin urmare, se obține relația anterioară.

Pentru a înțelege bine noțiunile prezentate în acest paragraf, ne vom referi la exemplele din paragraful 7.1:

Fie afirmația: gradul nodului i este k . Pentru exemplul 1., ea are semnificația că din orașul i pleacă (sosesc) k șosele, pentru exemplul 2., are semnificația că persoana i are k prieteni, pentru exemplul 3., are semnificația că țara i se învecinează cu k țări, iar pentru exemplul 4., are semnificația că pentru triunghiul i se cunosc k triunghiuri asemenea. Aici trebuie făcută observația că ar putea să existe și alte triunghiuri asemenea cu el, dar modul în care putem afla aceasta va fi tratat separat.

Fie afirmația: nodurile i și j sunt adiacente. Pentru exemplul 1., ea are semnificația că orașele i și j sunt unite printr-o șosea care nu trece prin alte orașe, pentru exemplul 2., are semnificația că persoanele i și j sunt prietene, pentru exemplul 3., are semnificația că țările i și j sunt vecine, iar pentru exemplul 4., are semnificația că triunghiurile i și j sunt asemenea.

Fie afirmația: nodul i este izolat. Pentru exemplul 1., înseamnă că nu există nici o șosea care leagă orașul i cu alt oraș, pentru exemplul 2., înseamnă că persoana i nu are nici un prieten, pentru exemplul 3., înseamnă că țara i nu se învecinează cu nici o țară (este situată pe o insulă), pentru exemplul 4., înseamnă că nu există nici un triunghi dintre celelalte $n-1$ triunghiuri care să fie asemenea cu triunghiul i .

7.3. Memorarea grafurilor

În acest paragraf, prezentăm principalele structuri de date prin care grafurile pot fi memorate în vederea prelucrării lor. De la început, precizăm faptul că vom alege o structură sau altă în funcție de²:

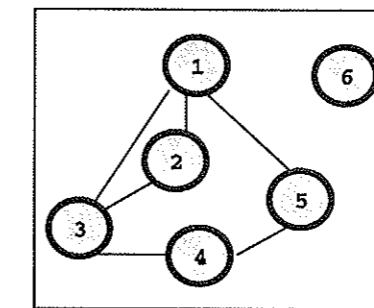
- a) algoritmul care prelucrează datele referitoare la graf;
- b) memoria internă pe care programul o are la dispoziție;
- c) dacă graful conține multe muchii sau nu.

Pentru fiecare structură de date pe care o vom folosi, vom avea câte o procedură (funcție) care citește datele respective. Toate aceste subprograme se

² Modul de alegere a structurii îl veți înțelege pe parcursul studiului acestui capitol.

găsesc grupate în unitatea de program **grafuri.pas** (pentru Pascal) și în **grafuri.cpp** (pentru C++). Vom fi astfel scutiți ca, pentru fiecare program pe care îl realizăm, să fim nevoiți să adăugăm liniile de cod necesare citirii și ne permite să ne concentrăm exclusiv asupra algoritmului pe care îl realizăm.

Ex: Toate subprogramele pe care le utilizăm citesc datele dintr-un fișier text, în care, pe prima linie vom scrie numărul de noduri (n), iar pe următoarele linii, câte o muchie (i, j) , ca în exemplul de mai jos, în care este prezentat un graf și liniile fișierului text care este citit pentru el:



Fișierul text:

6
1 2
1 3
1 5
2 3
3 4
4 5

Figura 7.3.
Exemplu de graf neorientat

Trecem la prezentarea structurilor prin care putem memora datele referitoare la un graf.

A. Memorarea grafului prin matricea de adiacență

$A_{n,n}$ – o matrice pătratică, unde elementele ei, $a_{i,j}$ au semnificația:

$$a_{i,j} = \begin{cases} 1, & \text{pentru } (i, j) \in E \\ 0, & \text{pentru } (i, j) \notin E \end{cases}$$

Pentru graful din figura 7.3, matricea de adiacență este prezentată în continuare:

$$A_{6,6} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Observații

1. Întrucât, din modul în care a fost definit graful, rezultă că nu există muchii de la un nod la el însuși, rezultă că elementele de pe **diagonala principală** rețin 0:

$$a_{i,i} = 0, \forall i \in \{1, 2, \dots, n\}.$$