

Ministerul Educației al Republicii Moldova

Anatol Gremalschi

INFORMATICĂ

Manual pentru clasa a **11**-a

Știința, 2014

CZU 004(075.3)

G 80

Elaborat conform curriculumului disciplinar în vigoare și aprobat prin Ordinul ministrului educației al Republicii Moldova (nr. 267 din 11 aprilie 2014). Editat din sursele financiare ale *Fondului Special pentru Manuale*.

Comisia de evaluare: *Gheorghe Curbet*, profesor școlar, grad didactic superior, Liceul Teoretic „Mihai Eminescu”, Bălți; *Arcadie Malearovici*, șef direcție, Centrul Tehnologiilor Informaționale și Comunicaționale în Educație, MET; *Varvara Vanovscaia*, profesor școlar, grad didactic superior, Liceul Teoretic „Vasile Alecsandri”, Chișinău

Recenzenți: *Gheorghe Căpățină*, doctor inginer, conferențiar universitar, Universitatea de Stat din Moldova; *Alexei Colîbneac*, maestru în arte, profesor universitar, Academia de Muzică, Teatru și Arte Plastice, Chișinău; *Tatiana Cartaleanu*, doctor în filologie, conferențiar universitar, Universitatea Pedagogică de Stat „Ion Creangă”, Chișinău; *Mihai Șleahțișchi*, doctor în psihologie și în pedagogie, conferențiar universitar, Universitatea Liberă Internațională din Moldova; *Valeriu Cabac*, doctor în științe fizico-matematice, conferențiar universitar, Universitatea de Stat „Alec Russo”, Bălți

Redactor: *Vasile Bahnaru*

Corectori: *Mariana Belenciuc, Elena Pistrui, Maria Cornesco*

Redactor tehnic: *Nina Duduciuc*

Machetare computerizată: *Anatol Andrițchi*

Copertă: *Vitalie Ichim*

Întreprinderea Editorial-Poligrafică *Știința*,

str. Academiei, nr. 3; MD-2028, Chișinău, Republica Moldova;

tel.: (+373 22) 73-96-16; fax: (+373 22) 73-96-27;

e-mail: prini@stiinta.asm.md; prini_stiinta@yahoo.com

www.stiinta.asm.md

DIFUZARE:

Republica Moldova: ÎM Societatea de Distribuție a Cărții *PRO-NOI*

str. Alba-Iulia, 75; MD-2051, Chișinău;

tel.: (+373 22) 51-68-17, 71-96-74; fax: (+373 22) 58-02-68;

e-mail: info@pronoi.md; www.pronoi.md

Toate drepturile asupra acestei ediții aparțin Întreprinderii Editorial-Poligrafice *Știința*.

Descrierea CIP a Camerei Naționale a Cărții

Gremalschi, Anatol

Informatică: Man. pentru clasa a 11-a/Anatol Gremalschi; Min. Educației al Rep. Moldova. – Ch.: Î.E.P. *Știința*, 2014 (Tipografia „BALACRON” SRL). – 192 p.

ISBN 978-9975-67-877-3

004(075.3)

CUPRINS

Conținuturi	Uma- nist	Real	Opțio- nal	Pag- na
Introducere				4
Capitolul 1. FUNCȚII ȘI PROCEDURI				5
1.1. Subprograme	•	•		5
1.2. Funcții	•	•		6
1.3. Proceduri	•	•		10
1.4. Domenii de vizibilitate	•	•		14
1.5. Comunicarea prin variabile globale	•	•		18
1.6. Efecte colaterale		•		20
1.7. Recursia		•		23
1.8. Sintaxa declarațiilor și apelurilor de subprograme	•	•		26
Capitolul 2. STRUCTURI DINAMICE DE DATE				30
2.1. Variabile dinamice. Tipul <i>referință</i>		•		30
2.2. Structuri de date		•		34
2.3. Liste unidirecționale		•		35
2.4. Prelucrarea listelor unidirecționale		•		40
2.5. Stiva		•		46
2.6. Cozi		•		51
2.7. Arbori binari		•		55
2.8. Parcurgerea arborilor binari		•		62
2.9. Arbori de ordinul m			•	67
2.10. Tipul de date <i>pointer</i>			•	72
Capitolul 3. METODE DE ELABORARE A PRODUSELOR PROGRAM				80
3.1. Programarea modulară			•	80
3.2. Testarea și depanarea programelor			•	87
3.3. Elemente de programare structurată			•	90
Capitolul 4. ANALIZA ALGORITMILOR				93
4.1. Complexitatea algoritmilor		•		93
4.2. Estimarea necesarului de memorie		•		95
4.3. Măsurarea timpului de execuție		•		100
4.4. Estimarea timpului cerut de algoritm		•		104
4.5. Complexitatea temporală a algoritmilor		•		109
Capitolul 5. TEHNICI DE ELABORARE A ALGORITMILOR				113
5.1. Iterativitate sau recursivitate		•		113
5.2. Metoda trierii		•		118
5.3. Tehnica <i>Greedy</i>		•		122
5.4. Metoda reluării		•		126
5.5. Metoda <i>desparte și stăpânește</i>			•	133
5.6. Programarea dinamică			•	140
5.7. Metoda <i>ramifică și mărginește</i>			•	144
5.8. Aplicațiile metodei <i>ramifică și mărginește</i>			•	147
5.9. Algoritmi exacti și algoritmi euristici			•	159
Capitolul 6. ALGORITMI DE REZOLVARE A UNOR PROBLEME MATEMATICE				170
6.1. Operații cu mulțimi			•	170
6.2. Analiza combinatorie			•	175
Capitolul 7. PROBLEME RECAPITULATIVE	•	•		183
Bibliografie				190

Dragi prieteni,

Manualul este elaborat în conformitate cu Curriculumul disciplinar de informatică și are drept scop însușirea de către elevi a cunoștințelor necesare pentru formarea culturii informaționale și dezvoltarea gândirii algoritmice. Cu ajutorul acestui manual veți studia funcțiile și procedurile limbajului PASCAL, structurile dinamice de date și metodele de elaborare a produselor program. De asemenea, veți studia cele mai răspândite tehnici de programare: trierea, tehnica Greedy, reluarea, metoda *desparte și stăpânește*, programarea dinamică, metoda *ramifică și mărginește*, algoritmi euristici. În manual sînt expuse metode de estimare a necesarului de memorie și a timpului cerut de algoritmi, recomandări ce vizează utilizarea recursiei și iterativității.

Modul de expunere a materialului este similar celui din manualele de informatică pentru clasele precedente. Mai întii se prezintă sintaxa și semantica unităților respective ale limbajului PASCAL, urmate de exemple de aplicare și recomandări pentru elaborarea de programe ce pot fi lansate pe calculator. În cazul tehnicilor de programare, sînt expuse noțiunile de bază și suportul matematic al tehnicii respective, urmate de modul de organizare a datelor, descrierea algoritmilor, elaborarea și depanarea programelor PASCAL. O atenție deosebită se acordă metodelor de implementare a tehnicilor de programare, interdependenței dintre performanțele calculatorului și complexitatea problemelor ce pot fi rezolvate cu ajutorul mijloacelor respective.

Implementarea tehnicilor de programare este ilustrată cu ajutorul mai multor probleme frecvent întâlnite în viața cotidiană și studiate în cadrul disciplinelor școlare din ciclul liceal. Totodată, în manual au fost incluse și probleme de o reală importanță practică, rezolvarea cărora este posibilă doar cu aplicarea calculatorului.

Fiind strîns legate de cunoștințele din alte domenii, temele din manual sînt axate pe metodele de rezolvare a problemelor ce necesită un volum foarte mare de calcul, evidențiindu-se rolul esențial al gândirii matematice în apariția și dezvoltarea informaticii. Exemplele, exercițiile și sarcinile individuale din manual vor contribui la perceperea adecvată a rolului și locului calculatorului, a influenței lui asupra dezvoltării matematicii, fizicii, chimiei, științelor socioumane. Pentru majoritatea temelor din manual au fost elaborate programe destinate instruirii asistate de calculator, fapt ce permite individualizarea procesului de predare-învățare, organizarea lecțiilor practice și dezvoltarea capacităților creative ale fiecărui elev.

În ansamblu, materialul inclus în *Manualul de informatică pentru clasa a XI-a* va contribui la dezvoltarea următoarelor competențe: analiza structurală a problemei; divizarea problemelor complexe în probleme mai simple și reducerea lor la cele deja rezolvate; estimarea complexității algoritmilor destinați soluționării problemelor propuse; utilizarea metodelor formale pentru elaborarea algoritmilor și scrierea programelor respective.

Evident, aceste calități sînt strict necesare nu numai viitorilor informaticieni, dar și fiecărui om cult care, la sigur, va trăi și va lucra într-un mediu bazat pe cele mai moderne tehnologii informaționale.

Autorul

FUNȚII ȘI PROCEDURI

1.1. Subprograme

E cunoscut faptul că o problemă complexă poate fi rezolvată prin divizarea ei într-un set de părți mai mici (subprobleme). Pentru fiecare parte se scrie o anumită secvență de instrucțiuni, denumită **subprogram**. Problema în ansamblu se rezolvă cu ajutorul programului principal, în care pentru rezolvarea subproblemelor se folosesc apelurile subprogramelor respective. Când în programul principal se întâlnește un apel, execuția continuă cu prima instrucțiune din programul apelat (fig. 1.1). Când se termină executarea instrucțiunilor din subprogram, se revine la instrucțiunea imediat următoare apelului din programul principal.

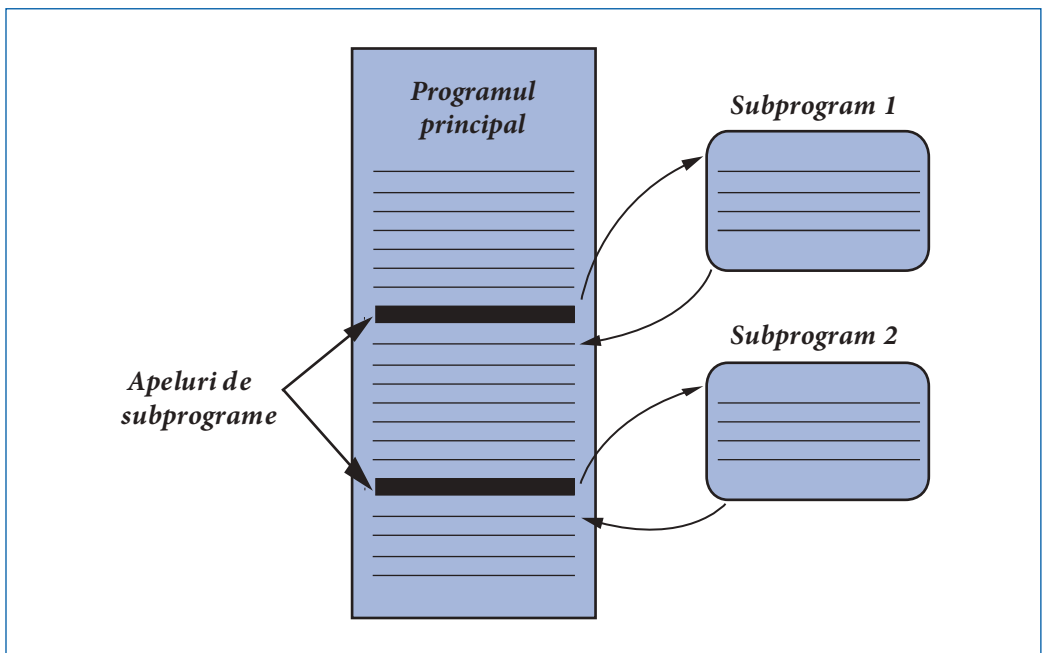


Fig. 1.1. Interacțiunea între program și subprogram

În limbajul PASCAL există două tipuri de subprograme, și anume, funcții și proceduri:

$\langle \text{Subprograme} \rangle ::= \{ \langle \text{Funcție} \rangle; \mid \langle \text{Procedură} \rangle; \}$

Funcțiile sînt subprograme care calculează și returnează o valoare. Limbajul PASCAL conține un set de funcții predefinite, cunoscute oricărui program: `sin`, `cos`, `eof` etc. În completare, programatorul poate defini funcții proprii, care se apelează în același mod ca și funcțiile-standard. Prin urmare, conceptul de funcție extinde noțiunea de **expresie** PASCAL.

Procedurile sînt subprograme care efectuează prelucrarea datelor comunicate în momentul apelului. Limbajul conține procedurile predefinite `read`, `readln`, `write`, `writeln` ș.a., studiate în clasele precedente. În completare, programatorul poate defini proceduri proprii, care se apelează în același mod ca procedurile-standard. Prin urmare, conceptul de procedură extinde noțiunea de **instrucțiune** PASCAL.

Subprogramele se definesc, în întregime, în partea declarativă a unui program. Evident, apelurile de funcții și proceduri se includ în partea executabilă a programului.

Un subprogram poate fi apelat chiar de el însuși, caz în care apelul este **recursiv**.

Întrebări și exerciții

- 1 Explicați termenii program principal și subprogram.
- 2 Cum interacționează programul și subprogramul?
- 3 Care este diferența dintre proceduri și funcții?
- 4 Cum se apelează o funcție? În care instrucțiuni ale limbajului pot apărea apeluri de funcții?
- 5 Cum se apelează o procedură?
- 6 Numiți tipul argumentului și tipul rezultatului furnizat de funcțiile predefinite `abs`, `chr`, `eof`, `eoln`, `exp`, `ord`, `sin`, `sqr`, `sqrt`, `pred`, `succ`, `trunc`.
- 7 Numiți tipul parametrilor actuali ai procedurilor `read` și `write`.
- 8 Ce prelucrări de date efectuează procedurile `read` și `write`?

1.2. Funcții

Textul PASCAL al unei **declarații de funcție** are forma:

```
function  $f(x_1, x_2, \dots, x_n) : t_r;$   
D;  
begin  
    ...  
     $f := e;$   
    ...  
end;
```

Prima linie este antetul funcției, format din:

f – numele funcției;

(x_1, x_2, \dots, x_n) – lista opțională de parametri formali reprezentînd argumentele funcției;

t_r – tipul rezultatului; acesta trebuie să fie numele unui tip simplu sau tip referință.

Antetul este urmat de **corpul funcției**, format din declarațiile locale opționale D și instrucțiunea compusă **begin ... end**.

Declarațiile locale sînt grupate în secțiunile (eventual vide) **label**, **const**, **type**, **var**, **function/procedure**.

Numele f al funcției apare cel puțin o dată în partea stîngă a unei instrucțiuni de atribuire care se execută: $f := e$. Ultima valoare atribuită lui f va fi întoarsă în programul principal.

În mod obișnuit, un parametru formal din lista (x_1, x_2, \dots, x_n) are forma:

$v_1, v_2, \dots, v_k : t_p$

unde v_1, v_2, \dots, v_k sînt identificatori, iar t_p este un nume de tip.

Utilizarea funcției f se specifică printr-un apel de forma:

$f(a_1, a_2, \dots, a_n)$

unde (a_1, a_2, \dots, a_n) este **lista de parametri actuali**. De obicei, parametrii actuali sînt expresii, valorile cărora sînt comunicate funcției. Corespondența între un parametru actual și parametrul formal se face prin poziția ocupată de aceștia în cele două liste. Parametrul actual trebuie să fie **compatibil** din punctul de vedere al atribuirii cu tipul parametrului formal.

Exemplu:

```
Program P97;  
{Declaraarea și utilizarea funcției Putere }  
type Natural=0..MaxInt;  
var a : real;  
    b : Natural;  
    c : real;  
    s : integer;  
    t : integer;  
    v : real;  
  
function Putere(x : real; n : Natural) : real;  
    {calcularea lui x la puterea n }  
var p : real;  
    i : integer;  
begin  
    p:=1;  
    for i:=1 to n do p:=p*x;  
    Putere:=p;  
end; { Putere }  
  
begin  
    a:=3.0;  
    b:=2;  
    c:=Putere(a, b);  
    writeln(a:10:5, b:4, c:10:5);
```

```

s:=2;
t:=4;
v:=Putere(s, t);
writeln(s:5, t:4, v:10:5);
readln;
end.

```

Funcția `Putere` are doi parametri formali: x de tipul `real` și n de tipul `Natural`. Funcția returnează o valoare de tipul `real`. În corpul funcției sînt declarate variabilele locale p și i .

La execuția apelului `Putere(a, b)` valorile 3.0 și 2 ale parametrilor actuali a, b se transmit parametrilor formali, respectiv, x și n . De menționat că tipul lui a coincide cu tipul lui x și tipul lui b coincide cu tipul lui n .

În cazul apelului `Putere(s, t)` tipul parametrilor actuali s, t nu coincide cu tipul parametrilor formali, respectiv, x și n . Totuși apelul este corect, întrucît tipurile respective sînt compatibile din punctul de vedere al atribuirii.

Întrebări și exerciții

- Se consideră următoarea declarație:

```

function Factorial(n : integer) : integer;
var p, i : integer;
begin
  p:=1;
  for i:=1 to n do p:=p * i;
  Factorial:=p;
end;

```

Numiți tipul parametrului formal și tipul rezultatului returnat de funcție. Precizați variabilele declarate în corpul funcției. Elaborați un program care afișează pe ecran valorile $n!$ pentru $n = 2, 3$ și 7 .

- În care loc al programului principal se includ declarațiile de funcții?
- Comentați programul ce urmează:

```

Program P98;
{ Eroare }
function Factorial(n : 0..7) : integer;
var p, i : integer;
begin
  p:=1;
  for i:=1 to n do p:=p*i;
  Factorial:=p;
end; { Factorial }
begin
  writeln(Factorial(4));
  readln;
end.

```


4 Se consideră antetul

```
function F(x : real; y : integer; z : char) : boolean;
```

Care din apelurile ce urmează sînt corecte:

a) F(3.18, 4, 'a')

e) F(3.18, 4, 4)

b) F(4, 4, '4')

f) F('3.18', 4, '4')

c) F(4, 4, 4)

g) F(15, 21, '3')

d) F(4, 3.18, 'a')

h) F(15, 21, 3)

5 Elaborați o funcție care calculează:

a) suma numerelor reale a, b, c, d ;

b) media numerelor întregi i, j, k, m ;

c) minimumul din numerele a, b, c, d ;

d) numărul de vocale într-un șir de caractere;

e) numărul de consoane într-un șir de caractere;

f) rădăcina ecuației $ax + b = 0$;

g) cel mai mic divizor al numărului întreg $n > 0$, diferit de 1;

h) cel mai mare divizor comun al numerelor naturale a, b ;

i) cel mai mic multiplu comun al numerelor naturale a, b ;

j) ultima cifră în notația zecimală a numărului întreg $n > 0$;

k) cîte cifre sînt în notația zecimală a numărului întreg $n > 0$;

l) cifra superioară în notația zecimală a numărului întreg $n > 0$;

m) numărul de apariții ale caracterului dat într-un șir de caractere.

6 Se consideră următoarele declarații:

```
const nmax=100;  
type Vector=array [1..nmax] of real;
```

Elaborați o funcție care calculează:

a) suma componentelor unui vector;

b) media componentelor vectorului;

c) componenta maximă;

d) componenta minimă.

7 Se consideră următoarele tipuri de date:

```
type Punct=record  
    x, y : real  
end;  
Segment=record  
    A, B : Punct  
end;
```

```

Triunghi=record
    A, B, C : Punct
end;
Dreptunghi=record
    A, B, C, D : Punct
end;
Cerc=record
    Centru : Punct;
    Raza : real
end;

```

Elaborați o funcție care calculează:

- lungimea segmentului;
- lungimea cercului;
- aria cercului;
- aria triunghiului;
- aria dreptunghiului.

- ⑧ Variabila *A* este introdusă prin declarația

```
var A : set of char;
```

Elaborați o funcție care returnează numărul de caractere din mulțimea *A*.

- ⑨ Elaborați o funcție care să calculeze diferența în secunde între două momente de timp date prin oră, minute și secunde.
- ⑩ Un triunghi este definit prin coordonatele vîrfurilor sale. Scrieți funcții care, pentru două triunghiuri date, să studieze dacă:
- au aceeași arie;
 - sînt asemenea;
 - primul este în interiorul celui de-al doilea.

1.3. Proceduri

Forma generală a textului unei declarații de procedură este:

```

procedure p(x1, x2, ..., xn);
    D;
begin
    ...
end;

```

În **antetul procedurii** apar:

p – numele procedurii;

*x*₁, *x*₂, ..., *x*_{*n*} – lista opțională de parametri formali;

În **corpul procedurii** sînt incluse:

D – declarațiile locale (opționale) grupate după aceleași reguli ca și în cazul funcțiilor;

begin ... end – instrucțiune compusă; ea nu conține vreo atribuire asupra numelui procedurii.

Procedura poate să întoarcă mai multe rezultate, dar nu prin numele ei, ci prin variabile desemnate special (cu prefixul **var**) în lista de parametri formali.

Parametrii din listă introduși prin declarații de forma

```
 $v_1, v_2, \dots, v_k : t_p$ 
```

se numesc *parametri-valoare*. Aceștia servesc pentru transmiterea de valori din programul principal în procedură.

Parametrii formali introduși în listă prin declarații de forma

```
var  $v_1, v_2, \dots, v_k : t_p$ 
```

se numesc *parametri-variabilă* și servesc pentru întoarcerea rezultatelor din procedură în programul principal.

Activarea unei proceduri se face printr-un apel de forma

```
 $p(a_1, a_2, \dots, a_n)$ 
```

unde a_1, a_2, \dots, a_n este lista de *parametri actuali*. Spre deosebire de funcție, apelul de procedură este o instrucțiune; aceasta se inserează în programul principal în locul în care sînt dorite efectele produse de execuția procedurii.

În cazul unui *parametru-valoare* drept parametru actual poate fi utilizată orice expresie de tipul respectiv, în particular o constantă sau o variabilă. Modificările parametrilor-valoare nu se transmit în exteriorul subprogramului.

În cazul unui *parametru-variabilă* drept parametri actuali pot fi utilizate numai variabile. Evident, modificările parametrilor în studiu vor fi transmise programului apelant.

Exemplu:

```
Program P99;  
{Declararea și utilizarea procedurii Lac }  
var a, b, c, t, q : real;  
  
procedure Lac(r : real; var l, s : real);  
{lungimea și aria cercului }  
{r - raza; l - lungimea; s - aria }  
const Pi=3.14159;  
begin  
  l:=2*Pi*r;  
  s:=Pi*sqr(r);  
end; {Lac }  
  
begin  
  a:=1.0;  
  Lac(a, b, c);  
  writeln(a:10:5, b:10:5, c:10:5);
```

```
Lac(3.0, t, q);
writeln(3.0:10:5, t:10:5, q:10:5);

readln;
end.
```

Procedura Lac are trei parametri formali: r, l și s. Parametrul r este un parametru-valoare, iar l și s sînt parametri-variabilă.

Execuția instrucțiunii Lac(a, b, c) determină transmiterea valorii 1.0 drept valoare a parametrului formal r și a locațiilor (adreselor) variabilelor b și c drept locații (adrese) ale parametrilor formali l și s. Prin urmare, secvența de instrucțiuni

```
a:=1.0;
Lac(a, b, c)
```

este echivalentă cu secvența

```
b:=2*Pi*1.0;
c:=Pi*sqr(1.0).
```

În mod similar, instrucțiunea

```
Lac(3.0, t, q)
```

este echivalentă cu secvența

```
t:=2*Pi*3.0;
q:=Pi*sqr(3.0).
```

Întrebări și exerciții

- ❶ Care este diferența dintre un *parametru-valoare* și un *parametru-variabilă*?
- ❷ Se consideră declarațiile:

```
var k, m, n : integer;
    a, b, c : real;
procedure P(i : integer; var j : integer;
           x : real; var y : real);
begin
  {...}
end.
```

Care din apelurile ce urmează sînt corecte?

a) P(k, m, a, b)

d) P(m, m, a, b)

b) P(3, m, a, b)

e) P(m, k, 6.1, b)

c) P(k, 3, a, b)

f) P(n, m, 6, b)

g) P(n,m,6,20)

i) P(i,i,i,i)

h) P(a,m,b,c)

j) P(a,a,a,a)

Argumentați răspunsul.

- 3 Comentați programul ce urmează:

```
Program P100;  
{Eroare }  
var a : real;  
    b : integer;  
procedure P(x : real; var y : integer);  
begin  
  {... }  
end; { P }  
begin  
  P(a, b);  
  P(0.1, a);  
  P(1, b);  
  P(a, 1);  
end.
```

- 4 Ce va afișa pe ecran programul ce urmează?

```
Program P101;  
{Parametru-valoare și parametru-variabilă }  
var a, b : integer;  
  
procedure P(x : integer; var y : integer);  
begin  
  x:=x+1;  
  y:=y+1;  
  writeln('x=', x, ' y=', y);  
end; {P }  
  
begin  
  a:=0;  
  b:=0;  
  P(a, b);  
  writeln('a=', a, ' b=', b);  
  readln;  
end.
```

Argumentați răspunsul.

- 5 Elaborați o procedură care:

- calculează rădăcinile ecuației $ax^2 + bx + c = 0$;
- radiază dintr-un șir caracterul indicat în apel;
- încadrează un șir de caractere între simbolurile „#”;

- d) ordonează componentele unui tablou **array** [1..100] **of** **real** în ordine crescătoare;
- e) ordonează componentele unui fișier **file of integer** în ordine descrescătoare;
- f) calculează și depune într-un tablou numerele prime mai mici decât un număr natural dat n .

6 Se consideră următoarele tipuri de date

```

type Data = record
    Ziua : 1..31;
    Luna : 1..12;
    Anul : integer;
end;
Persoana = record
    NumePrenume : string;
    DataNasterii : Data;
end;
ListaPersoane = array [1..50] of Persoana;

```

Elaborați o procedură care primește din programul principal o listă de persoane și restituie:

- a) persoanele născute în ziua z a lunii;
- b) persoanele născute în luna l a anului;
- c) persoanele născute în anul a ;
- d) persoanele născute pe data $z.l.a$;
- e) persoana cea mai în vîrstă;
- f) persoana cea mai tînăra;
- g) vîrsta fiecărei persoane în ani, luni, zile;
- h) lista persoanelor care au mai mult de v ani;
- i) lista persoanelor în ordine alfabetică;
- j) lista persoanelor ordonată conform datei nașterii;
- k) lista persoanelor de aceeași vîrstă (născuți în același an).

7 Elaborați o procedură care:

- a) creează o copie de rezervă a unui fișier text;
- b) exclude dintr-un fișier text liniile vide;
- c) numerotează liniile unui fișier text;
- d) concatenează două fișiere text într-unul singur;
- e) concatenează n fișiere text ($n > 2$) într-unul singur.

8 Vom numi *mari* numerele naturale care conțin mai mult de 20 de cifre semnificative. Să se definească un tip de date pentru numerele naturale mari și să se scrie proceduri care să adune și să scadă astfel de numere.

1.4. Domenii de vizibilitate

Corpul unui program sau subprogram se numește **bloc**. Deoarece subprogramele sînt incluse în programul principal și pot conține la rîndul lor alte subprograme, re-

zultă că blocurile pot fi **imbricate** (incluse unul în altul). Această imbricare de blocuri este denumită **structura de bloc** a programului PASCAL.

Într-o structură fiecărui bloc i se atașează câte un nivel de imbricare. Programul principal este considerat de nivel 0, un bloc definit în programul principal este de nivel 1. În general, un bloc definit în nivelul n este de nivelul $n + 1$.

Pentru exemplificare, în *figura 1.2* este prezentată structura de bloc a programului P105.

```
Program P105;                                     {nivel 0}
{ Structura de bloc a programului }
var a : real;
{1}

  procedure P(b : real);                           {nivel 1}
  var c : real;
  {2}

    procedure Q(d : integer);                       {nivel 2}
    {3}
    var c : char;
    {4}
    begin
      c:=chr(d);
      writeln('In procedura Q c=', c);
    end; {5}

  begin
    writeln('b=', b);
    c:=b+1;
    writeln('In procedura P c=', c);
    Q(35);
  end; {6}

  function F(x : real) : real;                     {nivel 1}
  begin
    f:=x/2;
  end;

begin
  a:=F(5);
  writeln('a=', a);
  P(a);
  readln;
end {7}.
```

Fig. 1.2. Structura de bloc a unui program PASCAL

De regulă, un bloc PASCAL include declarații de etichete, variabile, funcții, parametri ș.a.m.d. O declarație introduce un nume, care poate fi o etichetă sau un identificator. O declarație dintr-un bloc poate redefini un nume declarat în exteriorul lui. În consecință, în diferite părți ale programului unul și același nume poate desemna obiecte diferite.

Prin **domeniul de vizibilitate** al unei declarații se înțelege textul de program, în care numele introdus desemnează obiectul specificat de declarația în studiu. Domeniul de vizibilitate începe imediat după terminarea declarației și se sfârșește odată cu textul blocului respectiv. Deoarece blocurile pot fi imbricate, domeniul de vizibilitate nu este neapărat o porțiune continuă din textul programului. Domeniul de vizibilitate al unei declarații dintr-un bloc inclus **acoperă** domeniul de vizibilitate al declarației ce implică același nume din blocul exterior.

De exemplu, în programul P105 domeniul de vizibilitate al declarației **var a : real** este textul cuprins între punctele marcate {1} și {7}. Domeniul de vizibilitate al declarației **var c : real** este format din două fragmente de text cuprinse între {2}, {3} și {5}, {6}. Domeniul de vizibilitate al declarației **var c : char** este textul cuprins între {4} și {5}.

Cunoașterea domeniilor de vizibilitate ale declarațiilor este necesară pentru determinarea obiectului curent desemnat de un nume.

De exemplu, identificatorul *c* din instrucțiunea

```
c := chr(d)
```

a programului P105 desemnează o variabilă de tip *char*. Același identificator din instrucțiunea

```
c := b+1
```

desemnează o variabilă de tip *real*.

De reținut că declarația unui nume de funcție/procedură se consideră terminată la sfârșitul antetului. Prin urmare, domeniul de vizibilitate al unei astfel de declarații include și corpul funcției/procedurii respective. Acest fapt face posibil **apelul recursiv**: în corpul funcției/procedurii aceasta poate fi referită, fiind vizibilă. Evident, declarația unui parametru formal este vizibilă numai în corpul subprogramului respectiv.

De exemplu, domeniul de vizibilitate al declarației **procedure Q** este textul cuprins între punctele marcate {3} și {6}. Domeniul de vizibilitate al declarației **d : integer** este textul cuprins între {3} și {5}.

Întrebări și exerciții

- 1 Cum se determină domeniul de vizibilitate al unei declarații?
- 2 Determinați domeniile de vizibilitate ale declarațiilor **b : real** și **x : real** din programul P105 (fig. 1.2).
- 3 Precizați structura de bloc a programului ce urmează. Indicați domeniul de vizibilitate al fiecărei declarații și determinați obiectele desemnate de fiecare apariție a identificatorilor *c* și *x*.


```

Program P106;
  { Redefinirea constantelor }
  const c=1;

  function F1(x : integer) : integer;
  begin
    F1:=x+c;
  end; { F1 }

  function F2(c : real) : real;
  const x=2.0;
  begin
    F2:=x+c;
  end; { F2 }

  function F3(x : char) : char;
  const c=3;
  begin
    F3:=chr(ord(x)+c);
  end; { F3 }

  begin
    writeln('F1=', F1(1));
    writeln('F2=', F2(1));
    writeln('F3=', F3('1'));
    readln;
  end.

```

Ce va afișa pe ecran programul în studiu?

- ④ Determinați domeniile de vizibilitate ale identificatorilor P și F din programul P105 (fig. 1.2).
- ⑤ Comentați programul ce urmează:

```

Program P107;
  { Eroare }
  var a : real;

  procedure P(x : real);
  var a : integer;
  begin
    a:=3.14;
    writeln(x+a);
  end; { P }

  begin
    a:=3.14;
    P(a);
  end.

```

- ⑥ Cum se determină obiectul desemnat de apariția unui nume într-un program PASCAL?

1.5. Comunicarea prin variabile globale

Execuția unui apel de subprogram presupune transmiterea datelor de prelucrat funcției sau procedurii respective. După executarea ultimei instrucțiuni din subprogram, rezultatele produse trebuie întoarse în locul de apel. Cunoaștem deja că datele de prelucrat și rezultatele produse pot fi transmise prin parametri. Parametrii formali se specifică în antetul funcției sau procedurii, iar parametrii actuali – în locul apelului.

În completare la modul de transmitere a datelor prin parametri, limbajul PASCAL permite comunicarea prin variabile globale.

Orice variabilă este locală în subprogramul în care a fost declarată. O variabilă este **globală relativ la un subprogram** atunci când ea este declarată în programul sau subprogramul ce îl cuprinde fără să fie redeclarată în subprogramul în studiu. Întrucât variabilele globale sînt cunoscute atât în subprogram, cît și în afara lui, ele pot fi folosite pentru transmiterea datelor de prelucrat și returnarea rezultatelor.

Exemplu:

```
Program P108;
  {Comunicarea prin variabile globale }
var a,                {variabilă globală în P }
      b : real;        {variabilă globală în P, F }

procedure P;
var c : integer;     {variabilă locală în P }
begin
  c:=2;
  b:=a*c;
end; { P }

function F : real;
var a : 1..5;        {variabilă locală în F }
begin
  a:=3;
  F:=a+b;
end; { F }

begin
  a:=1;
  P;
  writeln(b);         {se afișează 2.0000000000E+00 }
  writeln(F);         {se afișează 5.0000000000E+00 }
  readln;
end.
```

Datele de prelucrat se transmit procedurii P prin variabila globală a. Rezultatul produs de procedură se returnează în blocul de apel prin variabila globală b. Valoarea

argumentului funcției F se transmite prin variabila globală b . Menționăm că variabila a este locală în F și nu poate fi folosită pentru transmiterea datelor în această funcție.

De obicei, comunicarea prin variabile globale se utilizează în cazurile în care mai multe subprograme prelucrează aceleași date. Pentru exemplificare amintim funcțiile cu argumente de tip *tablou*, procedurile care prelucrează tablouri și fișiere de angajați, persoane, elevi etc.

Întrebări și exerciții

- 1 Explicați termenii *variabilă globală* relativ la un subprogram și *variabilă locală* într-un subprogram.
- 2 Numiți variabilele globale și variabilele locale din programul P105 (fig. 1.2).
- 3 Poate fi oare o variabilă locală în același timp și o variabilă globală relativ la un subprogram?
- 4 Numiți variabilele globale și variabilele locale din programul ce urmează. Ce va afișa pe ecran acest program?

```
Program P109;
  {Comunicarea prin variabile globale }
  var a : integer;

  procedure P;
  var b, c, d : integer;

  procedure Q;
  begin
    c:=b+1;
  end; { Q }

  procedure R;
  begin
    d:=c+1;
  end; { R }
begin
  b:=a;
  Q;
  R;
  a:=d;
end; { P }

begin
  a:=1;
  P;
  writeln(a);
  readln;
end.
```

5 Se consideră declarațiile

```
Type Ora=0..23;  
Grade=-40..+40;  
Temperatura=array [Ora] of Grade;
```

Componentele unei variabile de tip `Temperatura` reprezintă temperaturile măsurate din oră în oră pe parcursul a 24 de ore. Elaborați o procedură care:

- a) indică maximumul și minimumul temperaturii;
- b) indică ora (orele) la care s-a înregistrat o temperatură maximă;
- c) înscrie ora (orele) la care s-a înregistrat o temperatură minimă într-un fișier `text`. Comunicarea cu procedurile respective se va face prin variabile globale.

6 Se consideră fișiere arbitrare de tip `text`. Elaborați o funcție care:

- a) returnează numărul de linii dintr-un fișier;
- b) calculează numărul de vocale dintr-un text;
- c) calculează numărul de cuvinte dintr-un text (cuvintele reprezintă șiruri de caractere separate prin spațiu sau sfârșit de linie);
- d) returnează lungimea medie a liniilor din text;
- e) calculează lungimea medie a cuvintelor din text;
- f) returnează numărul semnelor de punctuație din text.

Comunicarea cu funcțiile respective se va face prin variabile globale.

1.6. Efecte colaterale

Destinația unei funcții este să întoarcă ca rezultat o singură valoare. În mod obișnuit, argumentele se transmit funcției prin parametri-valoare, iar rezultatul calculat se returnează în locul de apel prin numele funcției. În completare, limbajul PASCAL permite transmiterea argumentelor prin variabile globale și parametri-variabilă.

Prin **efect colateral** se înțelege o atribuire (în corpul funcției) a unei valori la o variabilă globală sau la un parametru formal variabilă. Efectele colaterale pot influența în mod neașteptat execuția unui program și complică procesele de depanare.

Prezentăm în continuare exemple defectuoase de programare, care folosesc funcții cu efecte colaterale.

```
Program P110;  
{Efect colateral - atribuire la o variabilă globală}  
var a : integer; { variabilă globală }  
  
function F(x : integer) : integer;  
begin  
  F:=a*x;  
  a:=a+1;      {atribuire defectuoasă }  
end; { F }  
begin  
  a:=1;
```

```

writeln(F(1)); { se afișează 1 }
writeln(F(1)); { se afișează 2 }
writeln(F(1)); { se afișează 3 }
readln;
end.

```

În programul P110 funcția F returnează valoarea expresiei $a \cdot x$. Pe lângă aceasta însă, atribuirea $a := a + 1$ alterează valoarea variabilei globale a. În consecință, pentru una și aceeași valoare 1 a argumentului x funcția returnează rezultate diferite, fapt ce nu se încadrează în conceptul uzual de funcție.

```

Program P111;
{Efect colateral - atribuire la un parametru formal}
var a : integer;

function F(var x : integer) : integer;
  begin
    F:=2*x;
    x:=x+1; { atribuire defectuoasă }
  end; { F }

begin
  a:=2;
  writeln(F(a)); { se afișează 4 }
  writeln(F(a)); { se afișează 6 }
  writeln(F(a)); { se afișează 8 }
  readln;
end.

```

În programul P111 funcția F returnează valoarea expresiei $2 \cdot x$. Întrucît x este un parametru formal variabilă, atribuirea $x := x + 1$ schimbă valoarea parametrului actual din apel, și anume a variabilei a din programul principal. Faptul că apelurile textual identice $F(a)$, $F(a)$ și $F(a)$ returnează rezultate ce diferă poate crea confuzii în procesul depanării.

În cazul procedurilor, atribuirile asupra variabilelor globale produc efecte colaterale similare celor discutate pentru astfel de atribuirii la funcții. Întrucît mijlocul-standard de întoarcere de rezultate din procedură este prin parametri formali variabilă, atribuirile asupra unor astfel de parametri nu sînt considerate ca efecte colaterale.

Efectele colaterale introduc abateri de la procesul-standard de comunicare, prin care variabilele participante sînt desemnate explicit ca parametri formali în declarație și parametri actuali în apel. Consecințele efectelor colaterale se pot propaga în domeniul de vizibilitate al declarațiilor globale și pot interfera cu cele similare, produse la execuția altor proceduri și funcții. În astfel de condiții, utilizarea variabilelor globale devine riscantă. Prin urmare, la elaborarea programelor complexe se vor aplica următoarele recomandări:

1. Comunicarea funcțiilor cu mediul de chemare se va face prin transmiterea de date spre funcție prin parametri formali valoare și întoarcerea unui singur rezultat prin numele ei.
2. Comunicarea procedurilor cu mediul de chemare se va face prin transmiterea de date prin parametri formali valoare sau variabilă și întoarcerea rezultatelor prin parametri formali variabilă.
3. Variabilele globale pot fi folosite pentru transmiterea datelor în subprograme, însă valorile lor nu trebuie să fie schimbate de acestea.

Întrebări și exerciții

- ❶ Care este cauza efectelor colaterale? Ce consecințe pot avea aceste efecte?
- ❷ Precizați ce vor afișa pe ecran programele ce urmează:

```
Program P112;
{Efecte colaterale }
var a, b : integer;

function F(x : integer) : integer;
begin
  F:=a*x;
  b:=b+1;
end; { F }

function G(x : integer) : integer;
begin
  G:=b+x;
  a:=a+1;
end; { G }
begin
  a:=1; b:=1;
  writeln(F(1));
  writeln(G(1));
  writeln(F(1));
  writeln(G(1));
  readln;
end.
```

```
Program P113;
{Efecte colaterale }
var a : integer;
    b : real;

function F(var x : integer) : integer;
begin
  F:=x;
  x:=x+1;
end; { F }
```

```

procedure P(x,y:integer; var z:real);
begin
    z:=x/y;
end; { P }

begin
    a:=1;
    P(F(a), a, b);
    writeln(a, ' ', b);
    readln;
end.

```

```

Program P114;
    {Efecte colaterale }
var a, b : real;

procedure P(var x, y : real);
    {Interschimbarea valorilor variabilelor x, y }
begin
    a:=x;
    x:=y;
    y:=a;
end; { P }

begin
    a:=1; b:=2;
    P(a, b);
    writeln(a, b);
    a:=3; b:=4;
    P(a, b);
    writeln(a, b);
    readln;
end.

```

- ❸ Cum pot fi evitate efectele colaterale?

1.7. Recursia

Recursia se definește ca o situație în care un subprogram se autoapelează fie direct, fie prin intermediul altei funcții sau proceduri. Subprogramul care se autoapelează se numește *recursiv*.

De exemplu, presupunem că este definit tipul

```
type Natural = 0..MaxInt;
```

Funcția *factorial*

$$f(n) = \begin{cases} 1, & \text{dacă } n = 0; \\ n \cdot f(n-1), & \text{dacă } n > 0 \end{cases}$$

poate fi exprimată în PASCAL, urmînd direct definiția, în forma:

```
function F(n : Natural) : Natural;
begin
  if n=0 then F:=1
  else F:=n*F(n-1)
end; {F}
```

Efectul unui apel $F(7)$ este declanșarea unui lanț de apeluri ale funcției F pentru parametrii actuali 6, 5, ..., 2, 1, 0:

```
F(7) -> F(6) -> F(5) -> ... -> F(1) -> F(0).
```

Apelul $F(0)$ determină evaluarea directă a funcției și oprirea procesului repetitiv; urmează revenirile din apeluri și evaluarea lui F pentru 1, 2, ..., 6, 7, ultima valoare fiind întoarcerea în locul primului apel.

Funcția

$$fib(n) = \begin{cases} 0, & \text{dacă } n = 0; \\ 1, & \text{dacă } n = 1; \\ fib(n-1) + fib(n-2), & \text{dacă } n > 1 \end{cases}$$

are ca valori numerele lui *Fibonacci*. Urmînd definiția, obținem:

```
function Fib(n:Natural):Natural;
begin
  if n=0 then Fib:=0
  else if n=1 then Fib:=1
       else Fib:=Fib(n-1)+Fib(n-2)
end; {Fib}
```

Fiecare apel al funcției Fib pentru $n > 1$ generează două apeluri $Fib(n-1)$, $Fib(n-2)$ ș.a.m.d., de exemplu:

```
Fib(4) ->
  Fib(3), Fib(2) ->
    Fib(2), Fib(1), Fib(1), Fib(0) ->
      Fib(1), Fib(0).
```

Din exemplele în studiu se observă că recursia este utilă pentru programarea unor calcule repetitive. Repetiția este asigurată prin execuția unui subprogram care conține un apel la el însuși: cînd execuția ajunge la acest apel, este declanșată o nouă execuție ș.a.m.d.

Evident, orice subprogram recursiv trebuie să includă condiții de oprire a procesului repetitiv. De exemplu, în cazul funcției *factorial* procesul repetitiv se oprește cînd n ia valoarea 0; în cazul funcției Fib procesul se oprește cînd n ia valoarea 0 sau 1.

La orice apel de subprogram, în memoria calculatorului vor fi depuse următoarele informații:

- valorile curente ale parametrilor transmiși prin valoare;
- locațiile (adresele) parametrilor-variabilă;
- adresa de retur, adică adresa instrucțiunii ce urmează după apel.

Prin urmare, la apeluri recursive spațiul ocupat din memorie va crește rapid, riscând depășirea capacității de memorare a calculatorului. Astfel de cazuri pot fi evitate, înlocuind recursia prin iterație (instrucțiunile **for**, **while**, **repeat**). Pentru exemplificare prezentăm o formă nerecursivă a funcției *factorial*:

```
function F(n: Natural): Natural;
var i, p : Natural;
begin
  p:=1;
  for i:=1 to n do p:=p*i;
  F:=p;
end; {F}
```

Recursia este deosebit de utilă în cazurile în care elaborarea unor algoritmi nerecursivi este foarte complicată: traducerea programelor PASCAL în limbajul cod-mașină, grafica pe calculator, recunoașterea formelor ș.a.

Întrebări și exerciții

- 1 Cum se execută un subprogram recursiv? Ce informații se depun în memoria calculatorului la execuția unui apel recursiv?
- 2 Care este diferența dintre recursie și iterație?
- 3 Elaborați o formă nerecursivă a funcției lui *Fibonacci*.
- 4 Scrieți un subprogram recursiv care:
 - a) calculează suma $S(n) = 1 + 3 + 5 + \dots + (2n - 1)$;
 - b) calculează produsul $P(n) = 1 \times 4 \times 7 \times \dots \times (3n - 2)$;
 - c) inversează un șir de caractere;
 - d) calculează produsul $P(n) = 2 \times 4 \times 6 \times \dots \times 2n$.
- 5 Elaborați un program care citește de la tastatură numerele naturale m, n și afișează pe ecran valoarea funcției lui *Ackermann*:

$$a(m, n) = \begin{cases} n + 1, & \text{dacă } m = 0; \\ a(m - 1, 1), & \text{dacă } n = 0; \\ a(m - 1, a(m, n - 1)), & \text{dacă } m > 0 \text{ și } n > 0. \end{cases}$$

Calculați $a(0, 0)$, $a(1, 2)$, $a(2, 1)$ și $a(2, 2)$. Încercați să calculați $a(4, 4)$ și $a(10, 10)$. Explicați mesajele afișate pe ecran.

- 6 Se consideră declarația

```
type Vector=array [1..20] of integer;
```

Elaborați un subprogram recursiv care:

- a) afișează componentele vectorului pe ecran;
- b) calculează suma componentelor;

- c) inversează componentele vectorului;
- d) calculează suma componentelor pozitive;
- e) verifică dacă cel puțin o componentă a vectorului este negativă;
- f) calculează produsul componentelor negative;
- g) verifică dacă cel puțin o componentă a vectorului este egală cu un număr dat.

7 Elaborati o formă nerecursivă a funcției ce urmează:

```
function S(n:Natural):Natural;
begin
    if n=0 then S:=0
    else S:=n+S(n-1)
end; {S}
```

8 Scrieți o funcție recursivă care returnează valoarea true dacă șirul de caractere s este conform definiției

$\langle \text{Număr} \rangle ::= \langle \text{Cifră} \rangle \mid \langle \text{Cifră} \rangle \langle \text{Număr} \rangle$

Indicație. Forma unei astfel de funcții derivă din formula metalingvistică. Varianta nerecursivă

```
function N(s : string) : boolean;
var
    i : integer;
    p : boolean
begin
    p:=(s<>'');
    for i=1 to length(s) do
        p:=p and (s[i] in ['0'..'9']);
    N:=p;
end;
```

derivă din definiția

$\langle \text{Număr} \rangle ::= \langle \text{Cifră} \rangle \{ \langle \text{Cifră} \rangle \}$

9 Se consideră următoarele formule metalingvistice:

$\langle \text{Număr} \rangle ::= \langle \text{Cifră} \rangle \{ \langle \text{Cifră} \rangle \}$

$\langle \text{Semn} \rangle ::= + \mid -$

$\langle \text{Expresie} \rangle ::= \langle \text{Număr} \rangle \mid \langle \text{Expresie} \rangle \langle \text{Semn} \rangle \langle \text{Expresie} \rangle$

Scrieți o funcție recursivă care returnează valoarea true dacă șirul de caractere s este conform definiției unității lexicale $\langle \text{Expresie} \rangle$.

1.8. Sintaxa declarațiilor și apelurilor de subprograme

În general, definirea unei funcții se face cu ajutorul următoarelor formule metalingvistice:

$\langle \text{Funcție} \rangle ::= \langle \text{Antet funcție} \rangle ; \langle \text{Corp} \rangle$
 $\mid \langle \text{Antet funcție} \rangle ; \langle \text{Directivă} \rangle$
 $\mid \mathbf{function} \langle \text{Identificator} \rangle ; \langle \text{Corp} \rangle$

$\langle \text{Antet funcție} \rangle ::=$
function $\langle \text{Identificator} \rangle$ [$\langle \text{Listă parametri formali} \rangle$] : $\langle \text{Identificator} \rangle$

Diagramele sintactice sînt prezentate în figura 1.3.

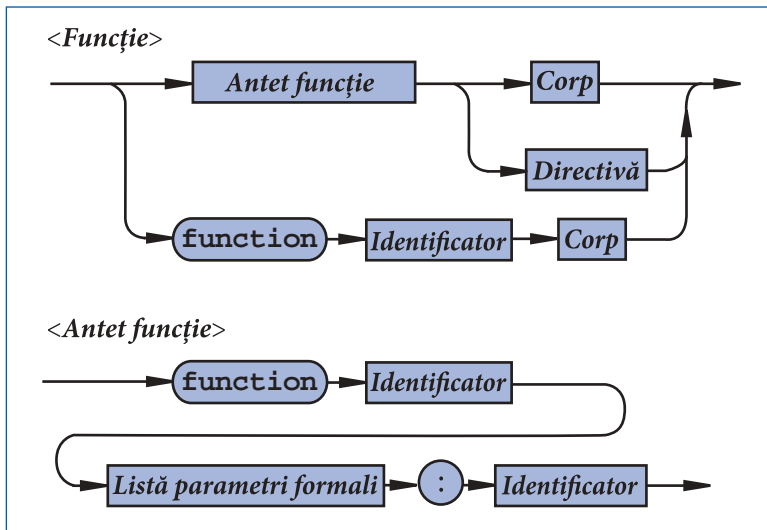


Fig. 1.3. Sintaxa declarațiilor de funcții

Procedurile se definesc cu ajutorul următoarelor formule:

$\langle \text{Procedură} \rangle ::= \langle \text{Antet procedură} \rangle ; \langle \text{Corp} \rangle \mid \langle \text{Antet procedură} \rangle ; \langle \text{Directivă} \rangle$
procedure $\langle \text{Identificator} \rangle ; \langle \text{Corp} \rangle$

$\langle \text{Antet procedură} \rangle :=$ **procedure** $\langle \text{Identificator} \rangle$ [$\langle \text{Listă parametri formali} \rangle$]

Diagramele sintactice sînt prezentate în figura 1.4.

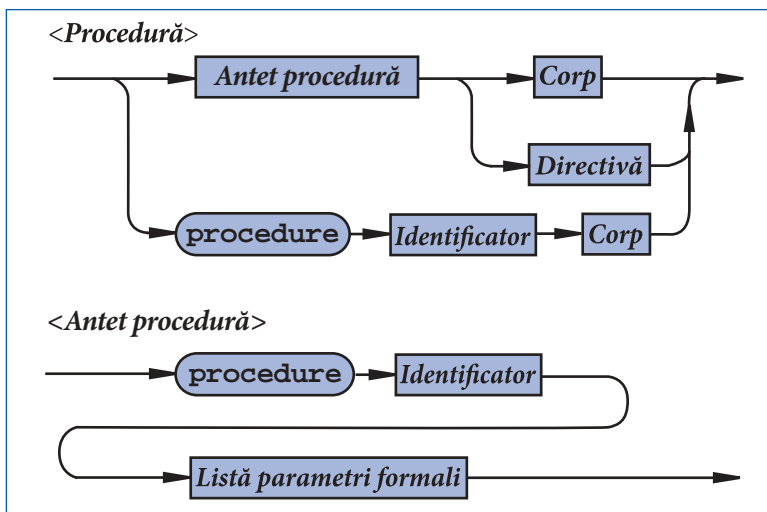


Fig. 1.4. Sintaxa declarațiilor de proceduri

Listele de parametri formali au următoarea sintaxă:

```

<Listă parametri formali> ::=
    (<Parametru formal> { ; <Parametru formal> } )
<Parametru formal> ::=
    [var] <Identificator> { , <Identificator> } : <Identificator>
    | <Antet funcție>
    | <Antet procedură>
    
```

Diagrama sintactică este prezentată în figura 1.5.

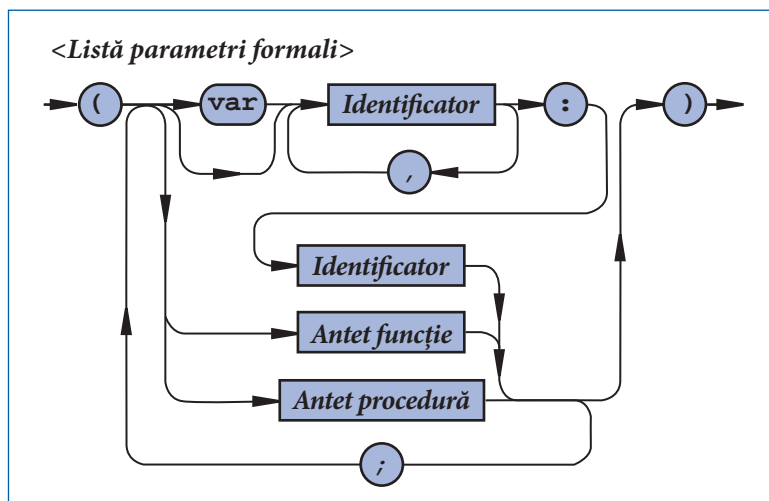


Fig. 1.5. Diagrama sintactică <Listă parametri formali>

Amintim că în lipsa cuvântului-cheie **var** identificatorii din listă specifică **parametrii-valoare**. Cuvântul **var** prefixează **parametrii-variabilă**. Antetul unei funcții (proceduri) din listă specifică un **parametru-funcție (procedură)**. În Turbo PASCAL astfel de parametri se declară explicit ca aparținând unui **tip procedural** și au forma parametrilor-valoare. Limbajul PASCAL extinde sensul uzual al noțiunii de funcție, permițând returnarea valorilor nu numai prin numele funcției, ci și prin parametri-variabilă.

Un **apel de funcție** are forma:

```

<Apel funcție> ::= <Nume funcție> [<Listă parametri actuali>]
    
```

iar o instrucțiune **apel de procedură**:

```

<Apel procedură> ::= <Nume procedură> [<Listă parametri actuali>]
    
```

Parametrii actuali se specifică cu ajutorul formulelor:

```

<Listă parametri actuali> ::= ( <Parametru actual> { , <Parametru actual> } )
    
```

```

<Parametru actual> ::= <Expresie> | <Variabilă> | <Nume funcție>
    | <Nume procedură>
    
```

Diagrama sintactică este prezentată în figura 1.6.

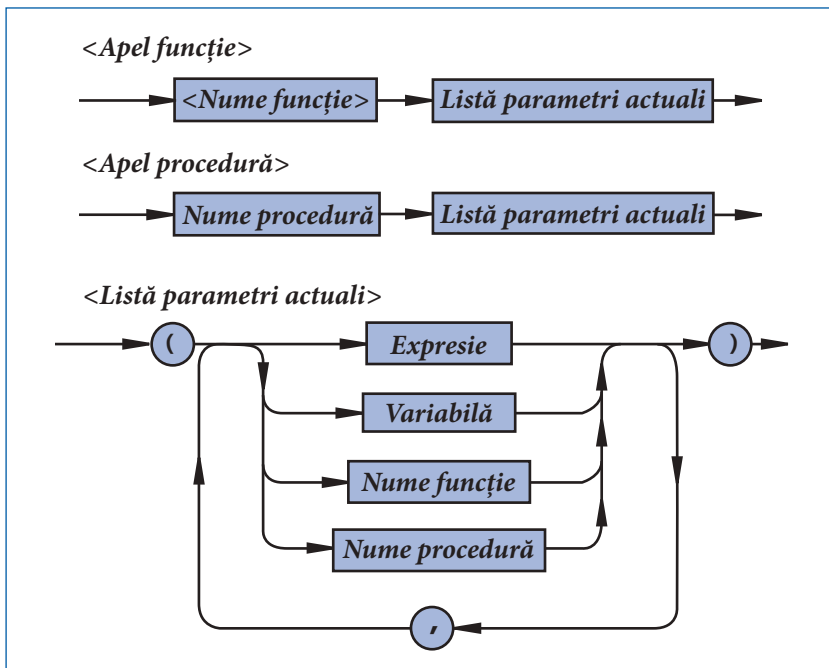


Fig. 1.6. Sintaxa apelurilor de funcții și proceduri

Correspondența între un parametru actual și parametrul formal se face prin poziția ocupată de aceștia în cele două liste.

În cazul unui **parametru-valoare** drept parametru actual poate fi utilizată orice expresie, în particular, o constantă sau o variabilă. Expresia respectivă trebuie să fie compatibilă din punctul de vedere al atribuirii cu tipul parametrului formal. Modificările parametrilor-valoare nu se transmit în exteriorul subprogramului.

În cazul unui **parametru-variabilă** drept parametri actuali pot fi utilizate numai variabile. Modificările parametrilor-variabilă se transmit în exteriorul subprogramului.

În cazul unui **parametru-funcție (procedură)** drept parametru actual poate fi utilizat orice nume de funcție (procedură), antetul căreia are forma specificată în lista parametrilor formali.

Întrebări și exerciții

- ❶ Când se utilizează declarațiile de forma `function <Identificator>; <Corp> ?`
- ❷ Indicați pe diagramele sintactice din figurile 1.3 și 1.5 drumurile care corespund declarațiilor de funcții din programul P106, paragraful 1.4.
- ❸ Care este diferența dintre un parametru-valoare și un parametru-variabilă?
- ❹ Indicați pe diagramele sintactice din figurile 1.4 și 1.5 drumurile care corespund declarațiilor de proceduri din programul P101, paragraful 1.3.
- ❺ Indicați pe diagramele sintactice din figurile 1.3–1.6 drumurile care corespund declarațiilor și apelurilor de subprograme din programul P105, paragraful 1.4.

STRUCTURI DINAMICE DE DATE

2.1. Variabile dinamice. Tipul *referință*

Variabilele declarate în secțiunea **var** a unui program sau subprogram se numesc **variabile statice**. Numărul variabilelor statice se stabilește în momentul scrierii programului și nu poate fi schimbat în timpul execuției. Există însă situații în care numărul necesar de variabile nu este cunoscut din timp.

De exemplu, presupunem că este necesară prelucrarea datelor referitoare la persoanele care formează un șir de așteptare (o coadă) la o casă de bilete. Lungimea cozii este nedefinită. De fiecare dată cum apare o persoană nouă, trebuie să se creeze o variabilă de tipul respectiv. După ce persoana pleacă, variabila corespunzătoare devine inutilă.

Variabilele care sînt create și eventual distruse în timpul execuției programului se numesc **variabile dinamice**.

Accesul la variabilele dinamice se face prin intermediul variabilelor de tip *referință*. De obicei, un tip *referință* se definește printr-o declarație de forma:

```
type  $T_r = ^T_b$ ;
```

unde T_r este numele tipului *referință*, iar T_b este tipul de bază. Semnul „^” se citește „adresă”. Evident, pot fi utilizate și tipuri referință anonime. Diagrama sintactică <Tip referință> este prezentată în figura 2.1.

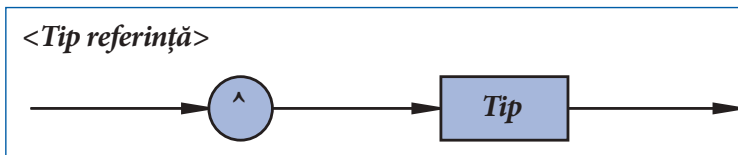


Fig. 2.1. Diagrama sintactică <Tip referință>

Mulțimea de valori ale unui tip de date *referință* constă din adrese. Fiecare adresă identifică o variabilă dinamică ce aparține tipului de bază. La această mulțime de adrese se mai adaugă o valoare specială, notată **nil** (zero), care nu identifică nicio variabilă.

Exemplu:

```
type AdresaInteger=^integer ;  
      AdresaChar=^char ;
```

```
var i : AdresaInteger ;
    r : ^real ;
    c : AdresaChar ;
```

Valoarea curentă a variabilei *i* va indica o variabilă dinamică de tipul *integer*. Într-un mod similar, variabilele de tip *referință* *r* și *c* identifică variabile de tipul *real* și, respectiv, *char*. Subliniem faptul că tipurile de date *AdresaInteger*, *AdresaChar* și tipul anonim *^real* sînt tipuri *referință* distincte.

Operațiile care se pot face cu valorile unui tip de date *referință* sînt = și <>. Valorile de tip *referință* nu pot fi citite de la tastatură și afișate pe ecran.

Crearea unei variabile dinamice se realizează cu procedura predefinită *new* (nou). Apelul acestei proceduri are forma

```
new(p)
```

unde *p* este o variabilă de tip *referință*.

Procedura alocă spațiu de memorie pentru variabila nou-creată și returnează adresa zonei respective prin variabila *p*. În continuare variabila dinamică poate fi accesată prin așa-zisa **dereperare**: numele variabilei de tip *referință* *p* este urmat de semnul de „^”. Dereperarea unei variabile de tip *referință* cu conținutul **nil** va declanșa o eroare de execuție.

Exemplu:

new(i) ; *i*^:=1 — crearea unei variabile dinamice de tipul *integer*; variabilei create *i* se atribuie valoarea 1;

new(r) ; *r*^:=2.0 — crearea unei variabile dinamice de tipul *real*; variabilei create *i* se atribuie valoarea 2.0;

new(c) ; *c*^:='*' — crearea unei variabile dinamice de tipul *char*; variabilei create *i* se atribuie valoarea '*'.

Subliniem faptul că variabila dinamică *p*^ obținută printr-un apel *new(p)* este distinctă de toate variabilele create anterior. Prin urmare, executarea instrucțiunilor

```
new(p) ; new(p) ; ... ; new(p)
```

conduce la crearea unui șir v_1, v_2, \dots, v_n de variabile dinamice. Numai ultima variabilă creată, v_n , este referită prin *p*^. Întrucît valorile variabilelor de tip *referință* reprezintă adresele anumitor zone din memoria internă a calculatorului, variabilele în studiu se numesc **indicatori de adresă**.

Distrugerea unei variabile dinamice și eliberarea zonei respective de memorie se realizează cu procedura predefinită *dispose* (a dispune). Apelul acestei proceduri are forma:

```
dispose(p)
```

unde *p* este o variabilă de tip *referință*.

Exemple:

```
dispose(i) ; dispose(r) ; dispose(c)
```

După executarea instrucțiunii `dispose(p)` valoarea variabilei de tip *referință* `p` este nedefinită.

Asupra variabilelor dinamice se pot efectua toate operațiile admise de tipul de bază.

Exemplu:

```
Program P117;
{Operații cu variabile dinamice }
type AdresaInteger=^integer;
var i, j, k : AdresaInteger;
    r, s, t : ^real;
begin
  {crearea variabilelor dinamice de tipul integer }
  new(i); new(j); new(k);
  {operații cu variabilele create }
  i^:=1; j^:=2;
  k^:=i^+j^;
  writeln(k^);
  {crearea variabilelor dinamice de tipul real }
  new(r); new(s); new(t);
  {operații cu variabilele create }
  r^:=1.0; s^:=2.0;
  t^:=r^/s^;
  writeln(t^);
  {distrugerea variabilelor dinamice }
  dispose(i); dispose(j); dispose(k);
  dispose(r); dispose(s); dispose(t);
  readln;
end.
```

Spre deosebire de variabilele statice, care ocupă zone de memorie stabilite de compilator, variabilele dinamice ocupă zone de memorie oferite de procedura `new`. Zonele respective sînt eliberate de procedura `dispose` și pot fi reutilizate pentru crearea unor variabile dinamice noi. Prin urmare, procedurile `new` și `dispose` asigură **alocarea (rezervarea) dinamică a memoriei**: spațiul de memorie este atribuit unei variabile dinamice numai pe durata existenței ei.

Numărul de variabile dinamice ce pot exista concomitent în timpul execuției unui program PASCAL depinde de tipul variabilelor și spațiul de memorie disponibil. În cazul în care tot spațiul de memorie este deja ocupat, apelul procedurii `new` va declanșa o eroare de execuție.

Exemplu:

```
Program P118;
{Eroare: depășirea capacității memoriei }
label l;
var i : ^integer;
```



```
begin
  l : new(i);
      goto l;
end.
```

Alocarea dinamică a memoriei necesită o atenție sporită din partea programatorului care este obligat să asigure crearea, distrugerea și referirea corectă a variabilelor dinamice.

Întrebări și exerciții

- ❶ Care este diferența dintre variabilele statice și variabilele dinamice?
- ❷ Cum se identifică variabilele dinamice?
- ❸ Indicați pe diagrama sintactică din *figura 2.1* drumurile care corespund declarațiilor de tipuri referință din programul P117.
- ❹ Se consideră declarațiile:

```
type AdresaReal = ^real;
var r : AdresaReal;
```

Precizați mulțimea de valori ale tipului de date *AdresaReal* și mulțimea de valori pe care le poate lua variabila dinamică *r*.

- ❺ Ce operații se pot efectua cu valorile unui tip de date *referință*? Cu variabilele dinamice?
- ❻ Se consideră declarațiile:

```
type AdresaTablou = ^array [1..10] of integer;
var t : AdresaTablou;
```

Precizați mulțimea de valori ale tipului de date *AdresaTablou* și mulțimea de valori pe care le poate lua variabila dinamică *t*.

- ❼ Comentați programul:

```
Program P119;
{Eroare }
var r, s : ^real;
begin
  r:=1; s:=2;
  writeln('r=', r, ' s=', s);
  readln;
end.
```

- ❽ Elaborați un program în care se creează două variabile dinamice de tipul *șir de caractere*. Atribuiți valori variabilelor create și afișați la ecran rezultatul concatenării șirurilor respective.
- ❾ Ce va afișa pe ecran programul ce urmează?

```
Program P120;
var i : ^integer;
```

```

begin
  new(i); i^:=1;
  new(i); i^:=2;
  new(i); i^:=3;
  writeln(i^);
  readln;
end.

```

⑩ Comentați programul:

```

Program P121;
{Eroare }
var i, j : ^integer;
begin
  new(i);
  i^:=1;
  dispose(i);
  new(j);
  j^:=2;
  dispose(j);
  writeln('i^=', i^, ' j^=', j^);
  readln;
end.

```

⑪ Explicați expresia *alocarea dinamică a memoriei*.

2.2. Structuri de date

O **structură de date** este formată din datele propriu-zise și relațiile dintre ele. În funcție de modul de organizare, o structură de date poate fi implicită sau explicită.

Tablourile, șirurile de caractere, articolele, fișierele și mulțimile studiate în capitolele precedente sînt **structuri implicite de date**. Relațiile dintre componentele acestor structuri sînt predefinite și nemodificabile. De exemplu, toate componentele unui șir de caractere au un nume comun, iar caracterul $s[i+1]$ este succesorul caracterului $s[i]$ în virtutea poziției ocupate.

Întrucît structura tablourilor, șirurilor de caractere, articolelor, mulțimilor și fișierelor nu se modifică în timpul execuției oricărui program sau subprogram, variabilele respective reprezintă **structuri statice de date**.

Folosind date cu structură implicită, putem rezolva reprezentativ o clasă limitată de probleme. În multe cazuri relațiile dintre componente nu numai că se modifică dinamic, dar în același timp pot deveni deosebit de complexe.

De exemplu, în cazul unui fir de așteptare la o casă de bilete relațiile dintre persoane se modifică: persoanele nou-sosite se așază la rînd; persoanele în criză de timp pleacă fără să-și mai procure bilete; persoanele care au plecat pentru un timp își păstrează rîndul ș.a.m.d. În cazul proiectării asistate de calculator a rețelelor de circula-

ție, stațiile, rutele, capacitatea de trafic ș.a. pot fi stabilite interactiv de către utilizator. În astfel de situații utilizarea datelor cu structură implicată devine nenaturală, dificilă și ineficientă.

Prin urmare, este necesară folosirea unor structuri de date în care relațiile dintre componente să fie reprezentate și prelucrate în mod explicit. Acest efect se poate obține atașând fiecărei componente o informație ce caracterizează relațiile acesteia cu alte date ale structurii. În cele mai multe cazuri, informația suplimentară, numită **informație de structură**, se reprezintă prin variabilele de tipul referință.

Structurile de date componentele cărora sînt create și eventual distruse în timpul execuției programului se numesc **structuri dinamice de date**. Structurile dinamice frecvent utilizate sînt: listele unidirecționale, listele bidirecționale, stivele, cozile, arborii ș.a.

O structură de date este **recursivă** dacă ea poate fi descompusă în date cu aceeași structură. Pentru exemplificare menționăm listele unidirecționale și arborii care vor fi studiați în paragrafele următoare.

Întrebări și exerciții

- 1 Explicați termenul *structură de date*. Dați exemple.
- 2 Care este diferența dintre structurile implicite și structurile explicite de date?
- 3 O structură de date este **omogenă** dacă toate componentele sînt de același tip. În caz contrar, structura de date este **eterogenă**. Dați exemple de structuri omogene și structuri eterogene de date.
- 4 Care este diferența dintre structurile statice și structurile dinamice de date?
- 5 Explicați termenul *structură recursivă de date*.

2.3. Liste unidirecționale

Listele unidirecționale sînt structuri explicite și dinamice de date formate din celule. Fiecare **celulă** este o variabilă dinamică de tipul **record** ce conține, în principal, două cîmpuri: cîmpul datelor și cîmpul legăturilor. **Cîmpul datelor** memorează informația prelucrabilă asociată celulei. **Cîmpul legăturilor** furnizează indicatorul de adresă corespunzător celulei la care se poate ajunge din celula curentă. Se consideră că orice celulă poate fi atinsă pornind de la o celulă privilegiată, numită **baza listei**.

Pentru exemplificare, în *figura 2.2* este prezentată o listă unidirecțională formată din 4 celule. Celulele conțin elementele *A*, *B*, *C* și *D*.

Datele necesare pentru crearea și prelucrarea unei liste unidirecționale pot fi definite prin declarații de forma:

```
type AdresaCelula=^Celula;  
Celula=record  
Info : string;
```

```

    Urm : AdresaCelula;
  end;
var P : AdresaCelula;

```

Informația utilă asociată unei celule se memorează în câmpul *Info*, iar adresa celulei următoare în câmpul *Urm*. Pentru simplificare se consideră că câmpul *Info* este de tipul *șir de caractere*. Ultima celulă din listă va avea în câmpul *Urm* valoarea **nil**. Adresa primei celule (adresa de bază) este memorată în variabila de tip referință *P* (fig. 2.2).

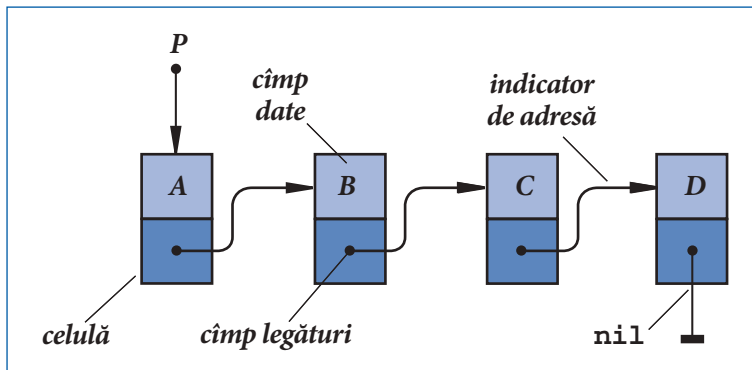


Fig. 2.2. Lista unidirecțională

Subliniem faptul că în definiția tipului referință *AdresaCelula* tipul de bază *Celula* încă nu este cunoscut. Conform regulilor limbajului PASCAL, acest lucru este posibil numai în cazul variabilelor dinamice, cu condiția ca tipul de bază să fie definit mai târziu în aceeași declarație.

O listă unidirecțională poate fi creată adăugând la vârful listei câte un element. Inițial lista în curs de construcție este vidă, adică nu conține nicio celulă.

Exemplu:

```

Program P122;
{Crearea listei unidirecționale A->B->C->D }
type AdresaCelula=^Celula;
     Celula=record
           Info : string;
           Urm : AdresaCelula;
     end;
var P,           {adresa de bază }
    R, V : AdresaCelula;
begin
  {1 - inițial lista este vidă }
  P:=nil;
  {2 - adăugarea celulei A }
  new(R);           {crearea unei celule }
  P:=R;           {inițializarea adresei de bază }

```

```

R^.Info:='A'; {încărcarea informației utile }
R^.Urm:=nil; {înscrierea indicatorului "sfârșit de listă" }
V:=R;      {memorarea adresei vârfului }
{3 - adăugarea celulei B }
new(R);    {crearea unei celule }
R^.Info:='B'; {încărcarea informației utile }
R^.Urm:=nil; {înscrierea indicatorului "sfârșit de listă" }
V^.Urm:=R; {crearea legăturii A -> B }
V:=R;      {actualizarea adresei vârfului }
{4 - adăugarea celulei C }
new(R);    {crearea unei celule }
R^.Info:='C'; {încărcarea informației utile }
R^.Urm:=nil; {înscrierea indicatorului "sfârșit de listă" }
V^.Urm:=R; {crearea legăturii B -> C }
V:=R;      {actualizarea adresei vârfului }
{5 - adăugarea celulei D }
new(R);    {crearea unei celule }
R^.Info:='D'; {încărcarea informației utile }
R^.Urm:=nil; {înscrierea indicatorului "sfârșit de listă" }
V^.Urm:=R; {crearea legăturii C -> D }
V:=R;      {actualizarea adresei vârfului }
{afișarea listei create }
R:=P;
while R<>nil do begin
    writeln(R^.Info);
    R:=R^.Urm
end;

readln;
end.

```

Procesul de construire a listei în studiu este prezentat în *figura 2.3*. Variabila *V* (adresa vârfului) din programul P122 reține adresa ultimei celule deja create pentru a-i poziționa indicatorul de adresă *V^.Urm*. Se procedează astfel pentru că în momentul în care completăm cîmpurile *R^.Info* și *R^.Urm* ale celulei curente încă nu se cunoaște adresa celulei ce urmează.

Listele cu un număr arbitrar de celule pot fi create și afișate cu ajutorul procedurilor respective din programul P123 :

```

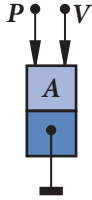
Program P123;
{ Crearea listelor unidirectionale}
type AdresaCelula=^Celula;
    Celula=record
        Info : string;
        Urm : AdresaCelula;
    end;

```

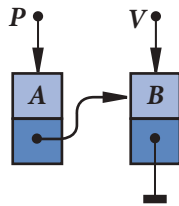
1 - inițial lista este vidă



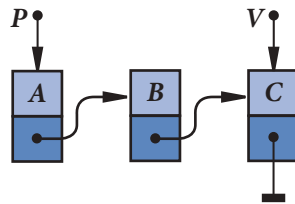
2 - adăugarea celulei A



3 - adăugarea celulei B



4 - adăugarea celulei C



5 - adăugarea celulei D

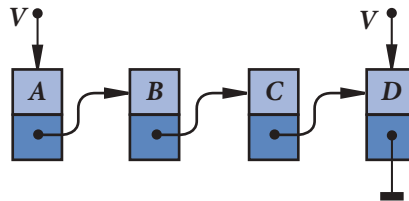


Fig. 2.3. Crearea listei unidirecționale

```

var p,q,r : AdresaCelula;
    s : string;
    i: integer;
procedure Creare;
begin
    p:=nil;
write('s='); readln(s);
    new(r); r^.Info:=s; r^.Urm:=nil;
    p:=r; q:=r;
    write('s=');
    while not eof do
        begin
            readln(s);write('s=');
            new(r); r^.Info:=s; r^.Urm:=nil;
            q^.Urm:=r; q:=r
        end;
end; { Creare }
procedure Afisare;
begin
    r:=p;
    while r<>nil do
        begin
            writeln(r^.Info);
            r:=r^.Urm;
        end;
readln;
end; { Afisare }

begin
    Creare;
    Afisare;
end.

```

Orice listă unidirecțională poate fi definită **recursiv** după cum urmează:

- a) o celulă este o listă unidirecțională;
- b) o celulă ce conține o legătură către o altă listă unidirecțională este o listă unidirecțională.

Pentru a sublinia faptul că listele unidirecționale sînt structuri recursive de date, declarațiile respective pot fi transcrise în forma:

```

type Lista=^Celula;
    Celula=record
        Info : string;
        Urm : Lista
    end;
var P : Lista;

```

Proprietățile listelor unidirecționale pot fi reproduse parțial prin memorarea elementelor respective într-un tablou unidimensional. De exemplu, datele din *figura 2.2* pot fi reprezentate în forma:

```
var L : array [1..4] of string;
    ...
    L[1] := 'A';
    L[2] := 'B';
    L[3] := 'C';
    L[4] := 'D';
    ...
```

Însă o astfel de reprezentare nu permite crearea și prelucrarea structurilor de date cu un număr arbitrar de elemente.

Întrebări și exerciții

- 1 Cum se definesc datele necesare pentru crearea unei liste?
- 2 Care este destinația câmpului datelor din componența unei celule? Care este destinația câmpului legăturilor? Ce informație se înscrie în acest câmp?
- 3 Scrieți un program care creează lista unidirecțională din *figura 2.2*, adăugînd cîte un element la baza listei.
- 4 Elaborați o procedură care schimbă cu locul două elemente din listă.
- 5 De la tastatură se citesc numere întregi diferite de zero. Se cere să se creeze două liste, una a numerelor negative, iar alta – a numerelor pozitive.
- 6 Prin ce se explică faptul că listele unidirecționale sînt structuri recursive de date?

2.4. Prelucrarea listelor unidirecționale

Operațiile frecvent utilizate în cazul listelor unidirecționale sînt:

- parcurgerea listei și prelucrarea informației utile asociate fiecărei celule;
- căutarea unui anumit element, identificat prin valoarea sa;
- includerea (inserarea) unui element într-un anumit loc din listă;
- excluderea (ștergerea) unui element dintr-o listă ș.a.

Presupunem că există o listă nevidă (*fig. 2.2*) definită prin declarațiile:

```
type AdresaCelula = ^Celula;
      Celula = record
          Info : string;
          Urm  : AdresaCelula
      end;
```

Variabila *P* indică adresa de bază a listei în studiu.

Parcurgerea listei se realizează conform următoarei secvențe de instrucțiuni:


```

R:=P; { poziționare pe celula de bază }
while R<>nil do
begin
  { prelucrarea informației din câmpul R^.Info }
  R:=R^.Urm; { poziționare pe celula următoare }
end;

```

Căutarea celulei ce conține elementul specificat de variabila Cheie este realizată de secvența:

```

R:=P;
while R^.Info<>Cheie do R:=R^.Urm;

```

Adresa celulei în studiu va fi reținută în variabila R.

Subliniem faptul că această secvență se execută corect numai în cazul în care lista include cel puțin o celulă ce conține informația căutată. În caz contrar, se ajunge la vârful listei, variabila R ia valoarea **nil**, iar dereperarea R^ provoacă o eroare de execuție. Pentru a evita astfel de erori, se utilizează secvența:

```

R:=P;
while R<>nil do
begin
  if R^.Info=Cheie then goto 1;
  R:=R^.Urm
end;
1: ...

```

Întrucât listele unidirecționale sînt structuri recursive de date, operația de căutare poate fi realizată și de un subprogram recursiv:

```

type Lista=^AdresaCelula;
      Celula=record;
              Info : string;
              Urm : Lista;
            end;
var P : Lista;
...
function Caut(P : Lista; Cheie : string):Lista;
begin
  if P=nil then Caut:=nil
  else
    if P^.Info=Cheie then Caut:=P
    else Caut:=Caut(P^.Urm, Cheie)
  end;
end;

```

Funcția Caut returnează adresa de bază a sublistei ce conține în prima celulă, dacă există, elementul specificat de parametrul Cheie.

Includerea celulei referite de indicatorul Q după celula referită de indicatorul R (fig. 2.4) este realizată de secvența de instrucțiuni:

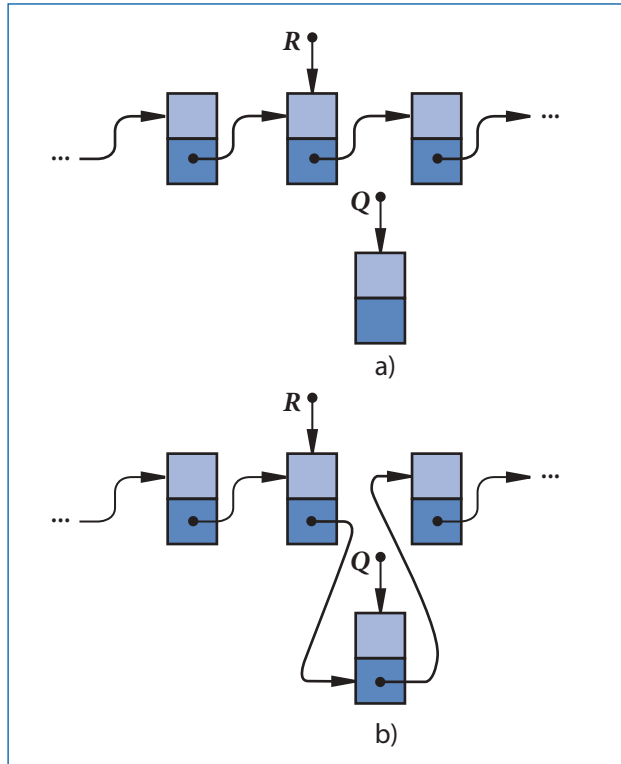


Fig. 2.4. Includerea unui element în listă:
 a – lista pînă la includere; b – lista după includere

```
Q^.Urm := R^.Urm;
R^.Urm := Q;
```

Excluderea celulei R din listă necesită aflarea adresei Q a celulei precedente și modificarea indicatorului de adresă $Q^.Urm$ (fig. 2.5):

```
Q := P;
while Q^.Urm <> R do Q := Q^.Urm;
Q^.Urm := R^.Urm;
```

Menționăm că includerea sau excluderea elementului din baza listei necesită actualizarea adresei de bază P.

Exemplu:

```
Program P124;
{Crearea și prelucrarea unei liste unidirecționale }
type AdresaCelula = ^Celula;
      Celula = record
          Info : string;
          Urm : AdresaCelula;
      end;
```

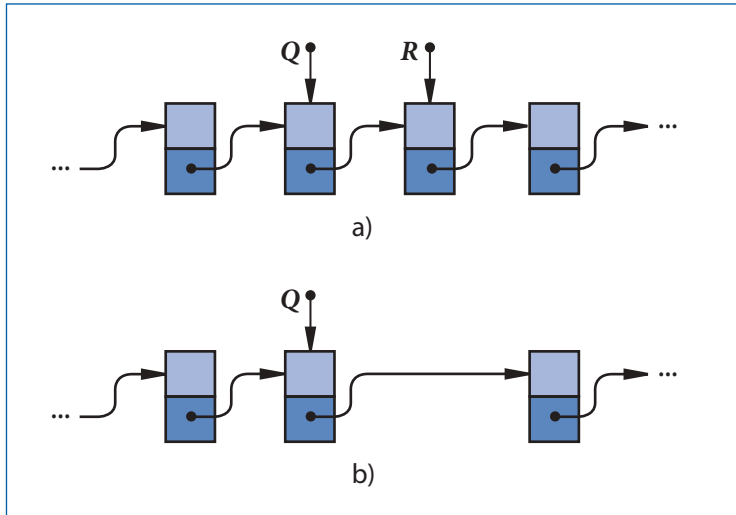


Fig. 2.5. Excluderea unui element din listă:
 a – lista pînă la excludere; b – lista după excludere

```

var P : AdresaCelula; {adresa de bază }
    c : char;
procedure Creare;
var R, V : AdresaCelula;
begin
  if P<>nil then writeln('Lista există deja')
  else begin
    writeln('Dați lista:');
    while not eof do
      begin
        new(R);
        readln(R^.Info);
        R^.Urm:=nil;
        if P=nil then begin P:=R; V:=R end
          else begin V^.Urm:=R; V:=R end;
      end;
    end;
  end; {Creare }

procedure Afis;
var R : AdresaCelula;
begin
  if P=nil then writeln('Lista este vidă')
  else begin
    writeln('Lista curentă:');
    R:=P;
    while R<>nil do
  
```

```

        begin
            writeln(R^.Info);
            R:=R^.Urm;
        end;
    end;
readln;
end; { Afis }

procedure Includ;
label 1;
var Q, R : AdresaCelula;
    Cheie : string;
begin
    new(Q);
    write('Dați elementul ce urmează');
    writeln(' să fie inclus:');
    readln(Q^.Info);
    write('Indicați elementul după care');
    writeln(' se va face includerea:');
    readln(Cheie);
    R:=P;
    while R<>nil do
        begin
            if R^.Info=Cheie then goto 1;
            R:=R^.Urm;
        end;
    1:if R=nil then begin
            writeln('Element inexistent');
            dispose(Q);
        end;
        else begin
            Q^.Urm:=R^.Urm;
            R^.Urm:=Q;
        end;
    end; { Includ }

procedure Exclud;
label 1;
var Q, R : AdresaCelula;
    Cheie : string;
begin
    write('Dați elementul ce urmează');
    writeln(' să fie exclus:');
    readln(Cheie);
    R:=P;
    Q:=R;

```

```

while R<>nil do
  begin
    if R^.Info=Cheie then goto 1;
    Q:=R;
    R:=R^.Urm;
  end;
1:if R=nil then writeln('Element inexistent')
  else begin
    if R=P then P:=R^.Urm else Q^.Urm:=R^.Urm;
    dispose(R);
  end;
end; { Exclud }
begin
  P:=nil; { inițial lista este vidă }
  repeat
    writeln('Meniu:');
    writeln('C - Crearea listei');
    writeln('I - Includerea elementului');
    writeln('E - Excluderea elementului');
    writeln('A - Afișarea listei la ecran');
    writeln('O - Oprirea programului');
    write('Opțiunea='); readln(c);
    case c of
      'C' : Creare;
      'I' : Includ;
      'E' : Exclud;
      'A' : Afis;
      'O' :
        else writeln('Opțiune necunoscută')
    end;
  until c='O';
end.

```

Procedura Creare formează o listă unidirecțională cu un număr arbitrar de celule. Informația utilă asociată fiecărei celule se citește de la tastatură. Procedura Afis afișează elementele listei la ecran. Procedura Includ citește de la tastatură elementul ce urmează să fie inclus și elementul după care se va face includerea. În continuare se caută celula ce conține elementul specificat, după care, dacă există, este inclusă celula nou-creată. Procedura Exclud caută și elimină, dacă există, celula ce conține elementul citit de la tastatură.

Întrebări și exerciții

- 1 Scrieți o funcție nerecursivă care returnează adresa vârfului listei unidirecționale. Transcrieți această funcție într-o formă recursivă.
- 2 Transcrieți procedurile Includ și Exclud din programul P124, fără a utiliza instrucțiunea goto.

- ③ Se consideră următoarele tipuri de date:

```
type AdresaCandidat=^Candidat;
      Candidat=record
          NumePrenume : string;
          NotaMedie : real;
          Urm : AdresaCandidat
      end;
```

Elaborați un program care:

- creează o listă unidirecțională cu componente de tipul `Candidat`;
 - afișează lista pe ecran;
 - exclue din listă candidatul care își retrage actele;
 - include în listă candidatul care depune actele;
 - afișează pe ecran candidații cu media mai mare de 7,5;
 - creează o listă suplimentară formată din candidații cu media mai mare de 9,0;
 - exclue din listă toți candidații cu media mai mică de 6,0.
- ④ Elaborați o procedură care:
- reordonează elementele listei unidirecționale conform unui anumit criteriu;
 - concatenează două liste unidirecționale;
 - descompune o listă în două liste;
 - selectează din listă elementele care corespund unui anumit criteriu.
- ⑤ Elementele listei unidirecționale sînt memorate într-un tablou unidirecțional. Elaborați procedurile necesare pentru:
- parcurgerea listei;
 - căutarea unui anumit element;
 - includerea unui element;
 - excluderea unui element.
- Care sînt avantajele și neajunsurile acestei reprezentări? Se consideră că lista va conține cel mult 100 de elemente.
- ⑥ Scrieți o funcție recursivă ce returnează numărul de celule dintr-o listă unidirecțională.
- ⑦ Scrieți un subprogram recursiv care exclue o anumită celulă din listă.
- ⑧ Prin **cuvînt** se înțelege orice secvență nevidă formată din literele alfabetului latin. Elaborați un program care formează lista cuvintelor întîlnite într-un fișier *text* și calculează numărul de apariții al fiecărui cuvînt. Examinați cazurile:
- cuvintele se includ în listă în ordinea primei lor apariții în text;
 - cuvintele se includ în listă în ordine alfabetică.
- Se consideră că literele mari și mici sînt identice.

2.5. Stiva

Prin **stivă** (în limba engleză *stack*) înțelegem o listă unidirecțională cu proprietatea că operațiile de introducere și extragere a elementelor se fac la un singur capăt

al ei. Poziția ocupată în stivă de ultimul element introdus poartă numele de **vîrf**. O stivă fără niciun element se numește **stivă vidă**.

Pentru exemplificare, în *figura 2.6* este prezentată o stivă care conține elementele A, B, C.

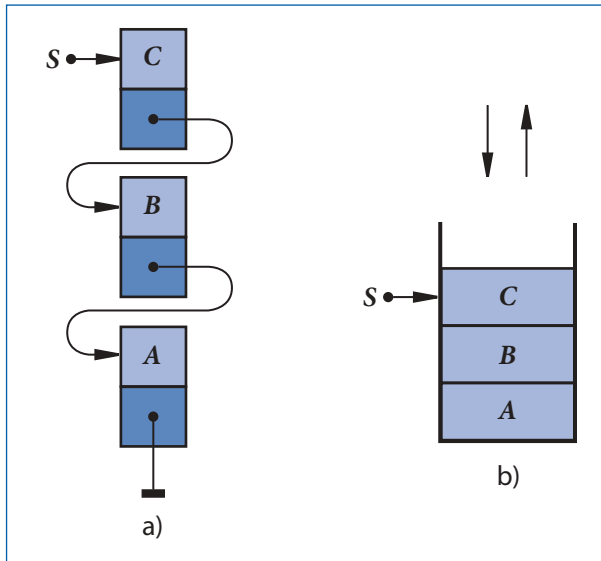


Fig. 2.6. Stiva:
a – reprezentarea detaliată; b – reprezentarea generalizată

Datele necesare pentru crearea și prelucrarea unei stive pot fi definite prin declarații de forma:

```

type AdresaCelula=^Celula;
      Celula=record
          Info : string;
          Prec : AdresaCelula
      end;
var S : AdresaCelula;
  
```

Adresa vârfului stivei este memorată în variabila de tip *referință* S. Adresa celei precedente din stivă este memorată în câmpul Prec.

Operația de introducere a unui element în stivă (*fig. 2.7*) este efectuată de secvența de instrucțiuni:

```

new(R); { crearea unei celule }
{încărcarea informației utile în câmpul R^.Info }
R^.Prec:=S; { crearea legăturii către celula precedentă
             din stivă }
S:=R; { actualizarea adresei vârfului }
  
```

unde R este o variabilă de tipul AdresaCelula.

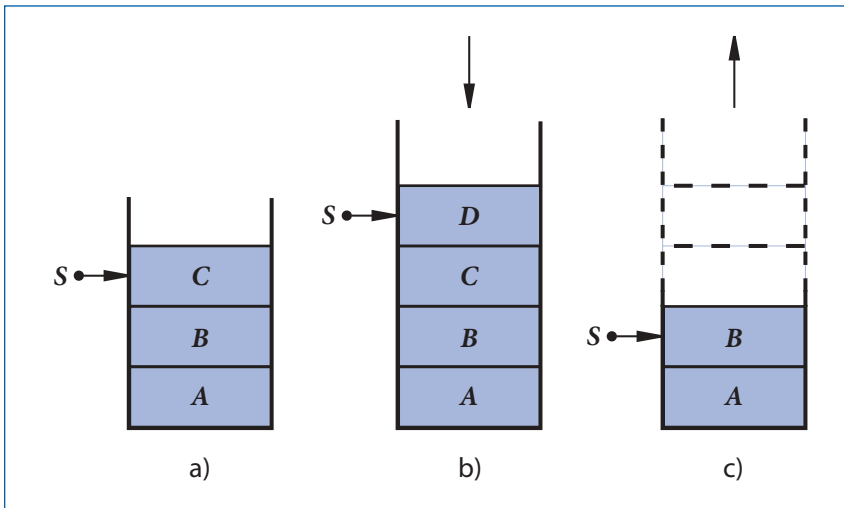


Fig. 2.7. Introducerea și extragerea elementelor din stivă:
a – stiva inițială; b – introducerea elementului D; c – extragerea elementelor D, C

Extragerea unui element din stivă (fig. 2.7) este efectuată de secvența:

```
R:=S; { memorarea adresei celulei extrase }
  { prelucrarea informației din câmpul R^.Info }
S:=S^.Prec; { eliminarea celulei din stivă }
dispose(R); { distrugerea celulei extrase }
```

Exemplu:

```
Program P127;
{Crearea și prelucrarea unei stive }
type AdresaCelula=^Celula;
      Celula=record
          Info : string;
          Prec : AdresaCelula;
      end;
var S : AdresaCelula; {adresa vârfului }
    c : char;
procedure Introduc;
var R : AdresaCelula;
begin
    new(R);
    write('Dați elementul ce urmează');
    writeln(' să fie introdus:');
    readln(R^.Info);
    R^.Prec:=S;
    S:=R;
end; { Includ }
```



```

procedure Extrag;
var R : AdresaCelula;
begin
    if S=nil then writeln('Stiva este vidă')
        else begin
            R:=S;
            write('Este extras');
            writeln(' elementul:');
            writeln(R^.Info);
            S:=S^.Prec;
            dispose(R);
        end;
end; { Extrag }

procedure Afis;
var R : AdresaCelula;
begin
    if S=nil then writeln('Stiva este vidă')
        else begin
            writeln('Stiva include elementele:');
            R:=S;
            while R<>nil do begin
                writeln(R^.Info);
                R:=R^.Prec;
            end;
        end;
    readln;
end; { Afis }
begin
    S:=nil; { inițial stiva este vidă }
    repeat
        writeln('Meniu:');
        writeln('I - Introducerea elementului');
        writeln('E - Extragerea elementului');
        writeln('A - Afișarea stivei pe ecran');
        writeln('O - Oprirea programului');
        write('Opțiunea='); readln(c);
        case c of
            'I' : Introduc;
            'E' : Extrag;
            'A' : Afis;
            'O' :
                else writeln('Opțiune necunoscută')
        end;
    until c='O';
end.

```

Stivele mai poartă și numele de **liste LIFO** (*last in, first out* – ultimul element care a intrat în stivă va fi primul care va ieși din ea) și sînt frecvent utilizate pentru alocarea dinamică a memoriei în cazul procedurilor și funcțiilor recursive. Evident, stivele pot fi simulate utilizînd tablourile unidimensionale **array** [1..n] of ..., însă o astfel de reprezentare este limitată de cele n componente ale tablourilor.

Întrebări și exerciții

- 1 Care este ordinea de introducere și extragere a datelor din stivă?
- 2 De la tastatură se citesc mai multe numere naturale. Afișați numerele în studiu pe ecran în ordinea inversă citirii.
- 3 În *figura 2.8* este reprezentată schema de manevrare a vagoanelor de tren într-un depou. Elaborați un program care citește de la tastatură și afișează pe ecran datele despre fiecare vagon intrat sau ieșit din depou. Datele în studiu includ:
 - numărul de înmatriculare (**integer**);
 - stația de înmatriculare (**string**);
 - anul fabricării (1960 . . 2000);
 - tipul vagonului (**string**);
 - capacitatea de încărcare (**real**);
 - proprietarul vagonului (**string**).

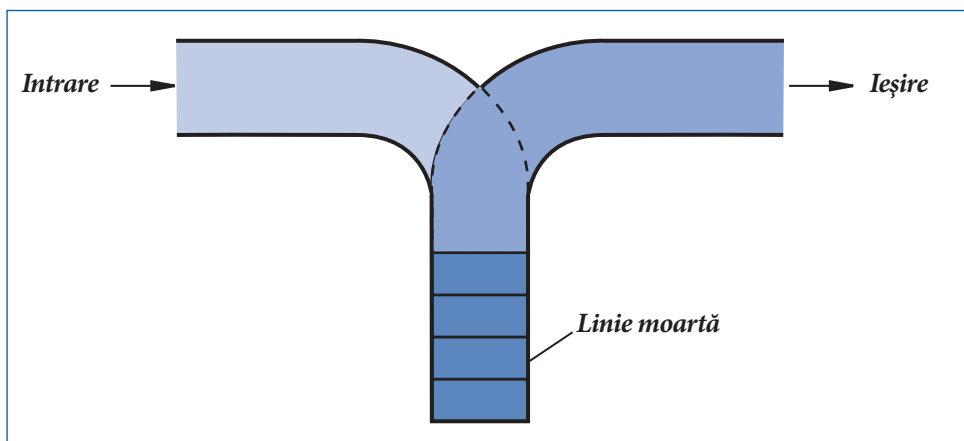


Fig. 2.8. Schema de manevrare a vagoanelor de tren

- 4 Colectivele temporare de muncă sînt formate și desființate în ordinea „ultimul angajat va fi primul care va fi concediat”. Elaborați un program care citește de la tastatură și afișează pe ecran datele despre fiecare persoană angajată sau concediată. Datele în studiu includ:
 - numele (**string**);
 - prenumele (**string**);
 - anul nașterii (1930 . . 1985);
 - data angajării (ziua, luna, anul).
- 5 Se consideră șiruri finite de caractere formate din parantezele (,), [,], {, }. Un șir este **corect** numai atunci cînd el poate fi construit cu ajutorul următoarelor reguli:
 - a) șirul vid este corect;

b) dacă A este un șir corect, atunci (A) , $[A]$ și $\{A\}$ sînt șiruri corecte;

c) dacă A și B sînt șiruri corecte, atunci AB este un șir corect.

De exemplu, șirurile $()$, $[\]$, $\{\}$, $[(\)]$, $((([\]))([\]))$ sînt corecte, iar șirurile $([$, $(\)\{\}$, $(\)$ nu sînt corecte. Elaborați un program care verifică dacă șirul citit de la tastatură este corect.

Indicație. Problema poate fi rezolvată printr-o singură parcurgere a șirului supus verificării. Dacă caracterul curent este $($, $[$ sau $\{$, el este depus în stivă. Dacă vîrfurile și caracterul curent formează una din perechile $(\)$, $[\]$ sau $\{\}$, paranteza respectivă este scoasă din stivă. În cazul unui șir corect, după examinarea ultimului caracter din șir stiva rămîne vidă.

- 6 Elemente stivei sînt memorate într-un tablou unidimensional. Elaborați procedurile necesare pentru introducerea și extragerea elementelor din stivă. Care sînt avantajele și neajunsurile acestei reprezentări? Se consideră că stiva conține cel mult 100 de elemente.

2.6. Cozi

Prin **coadă** (în engleză *queue*) înțelegem o listă unidirecțională în care toate introducerile se efectuează la unul din capete, iar extragerile se efectuează la celălalt capăt. O coadă fără niciun element se numește **coadă vidă**.

Pentru exemplificare, în figura 2.9 este prezentată o coadă care conține elementele A , B , C .

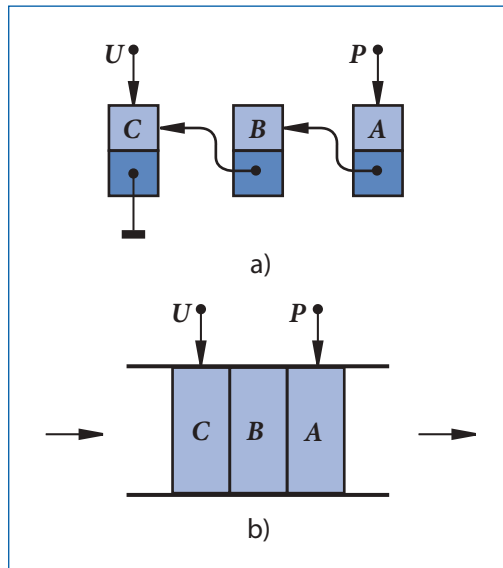


Fig. 2.9. Coada:

a – reprezentarea detaliată; b – reprezentarea generalizată

Datele necesare pentru crearea și prelucrarea unei cozi pot fi definite prin declarații de forma:

```
type AdresaCelula=^Celula;  
      Celula=record  
          Info : string;
```

```

    Urm : AdresaCelula
end;
var P, U : AdresaCelula;

```

Adresa primului element din coadă este memorată în variabila de tip referință P, iar adresa ultimului element în variabila U. Adresa celulei următoare din coadă este memorată în câmpul Urm.

Operația de introducere a unui element (fig. 2.10) este efectuată de secvența de instrucțiuni:

```

new(R); {crearea unei celule }
{încărcarea informației utile în câmpul R^.Info }
R^.Urm:=nil; { înscrierea indicatorului "ultimul element" }
U^.Urm:=R;   { adăugarea celulei la coadă }
U:=R;       { actualizarea adresei ultimei celule }

```

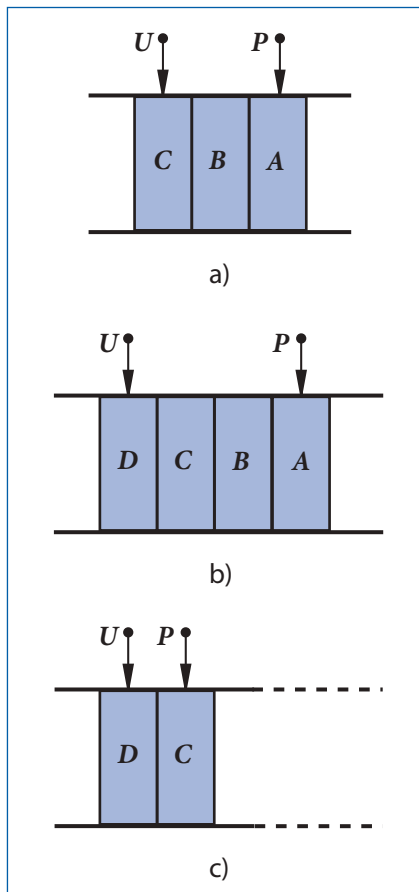


Fig. 2.10. Introducerea și extragerea elementelor din coadă:
a – coada inițială; b – introducerea elementului D;
c – extragerea elementelor A, B

Extragerea unui element din coadă (fig. 2.10) este efectuată de secvența:

```
R:=P; { memorarea adresei primei celule }  
{prelucrarea informației din câmpul R^.Info }  
P:=P^.Urm; {eliminarea primei celule }  
dispose(R); {distrugerea celulei extrase }
```

Exemplu:

```
Program P128;  
{Crearea și prelucrarea unei cozi }  
type AdresaCelula=^Celula;  
    Celula=record  
        Info : string;  
        Urm : AdresaCelula;  
    end;  
var P,                {adresa primului element }  
    U : AdresaCelula; {adresa ultimului element }  
    c : char;  
  
procedure Introduc;  
var R : AdresaCelula;  
begin  
    new(R);  
    write('Dați elementul ce urmează');  
    writeln(' să fie introdus:');  
    readln(R^.Info);  
    R^.Urm:=nil;  
    if P=nil then begin P:=R; U:=R end  
        else begin U^.Urm:=R; U:=R end;  
end; { Introduc }  
  
procedure Extrag;  
var R : AdresaCelula;  
begin  
    if P=nil then writeln('Coadă este vidă')  
        else begin  
            R:=P;  
            write('Este extras');  
            writeln(' elementul:');  
            writeln(R^.Info);  
            P:=P^.Urm;  
            dispose(R);  
        end;  
end; { Extrag }
```

```

procedure Afis;
var R : AdresaCelula;
begin
  if P=nil then writeln('Coadă este vidă')
  else begin
    write('Coadă include');
    writeln(' elementele:');
    R:=P;
    while R<>nil do
      begin
        writeln(R^.Info);
        R:=R^.Urm;
      end;
    end;
  end;

  readln;
end; { Afis }

begin
  P:=nil; U:=nil; { inițial coada este vidă }
  repeat

    writeln('Meniu:');
    writeln('I - Introducerea elementului;');
    writeln('E - Extragerea elementului;');
    writeln('A - Afișarea cozii la ecran;');
    writeln('O - Oprirea programului');
    write('Opțiunea='); readln(c);
    case c of
      'I' : Introduc;
      'E' : Extrag;
      'A' : Afis;
      'O' :
        else writeln('Opțiune necunoscută')
    end;
  until c='O';
end.

```

Cozile mai poartă numele de **liste FIFO** (*first in, first out* – primul element intrat în coadă va fi primul ieșit din coadă). Menționăm că simularea cozilor cu ajutorul tablourilor unidimensionale este inefficientă din cauza migrării elementelor cozii spre ultima componentă a tabloului.

Întrebări și exerciții

- ❶ Elaborați o funcție care returnează numărul elementelor unei cozi.
- ❷ Avioanele care solicită aterizarea pe o anumită pistă a unui aeroport formează un fir de așteptare. Elaborați un program care citește de la tastatură și afișează pe ecran datele

despre fiecare avion care solicită aterizarea și avionul care aterizează. Datele în studiu includ:

- numărul de înmatriculare (`integer`);
- tipul avionului (`string`);
- numărul rutei (`integer`).

- ❸ Prin **coadă cu priorități** vom înțelege o coadă în care elementul de introdus se inserează nu după ultimul element al cozii, ci înaintea tuturor elementelor cu o prioritate mai mică. Prioritățile elementelor se indică prin numere întregi. Elaborați un program care:
- crează o coadă cu priorități;
 - introduce în coadă elementele specificate de utilizator;
 - extrage elementele din coadă;
 - afișează coada cu priorități pe ecran.

2.7. Arbori binari

Prin **nod** se înțelege o variabilă dinamică de tipul **record** care conține un câmp destinat memorării informațiilor utile și doi indicatori de adresă.

Arborele binar se definește recursiv după cum urmează:

- un nod este un arbore binar;
- un nod ce conține legături către alți doi arbori binari este un arbore binar.

Prin convenție, **arborele vid** nu conține niciun nod. Pentru exemplificare, în *figura 2.11* este prezentat un arbore binar nodurile căruia conțin informația utilă *A, B, C, D, E, F, G, H, I, J*. Datele necesare pentru crearea și prelucrarea unui arbore binar pot fi definite prin declarații de forma:

```
type AdresaNod=^Nod;
      Nod=record
          Info : string;
          Stg, Dr : AdresaNod
      end;
var T : AdresaNod;
```

Pentru a sublinia faptul că arborii binari sînt **structuri recursive** de date, declarațiile în studiu pot fi transcrise în forma:

```
type Arbore=^Nod;
      Nod=record
          Info : string;
          Stg, Dr : Arbore;
      end;
var T : Arbore;
```

Nodul spre care nu este îndreptată nicio legătură se numește **rădăcină**. Adresa rădăcinii se memorează în variabila de tip *referință* T. În cazul unui arbore vid T=**nil**.

Cei doi arbori conectați la rădăcină se numesc **subarborul stâng** și **subarborul drept**. Adresa subarborului stâng se memorează în câmpul *Stg*, iar adresa subarborului drept – în câmpul *Dr*.

Nivelul unui nod este, prin convenție, 0 pentru nodul-rădăcină și $i + 1$, pentru nodul conectat la un nod de nivelul i . În mod obișnuit, în reprezentarea grafică a unui arbore binar nodurile se desenează pe niveluri: rădăcina se află pe nivelul 0, vîrfurile conectate la rădăcină – pe nivelul 1 ș.a.m.d. (fig. 2.11).

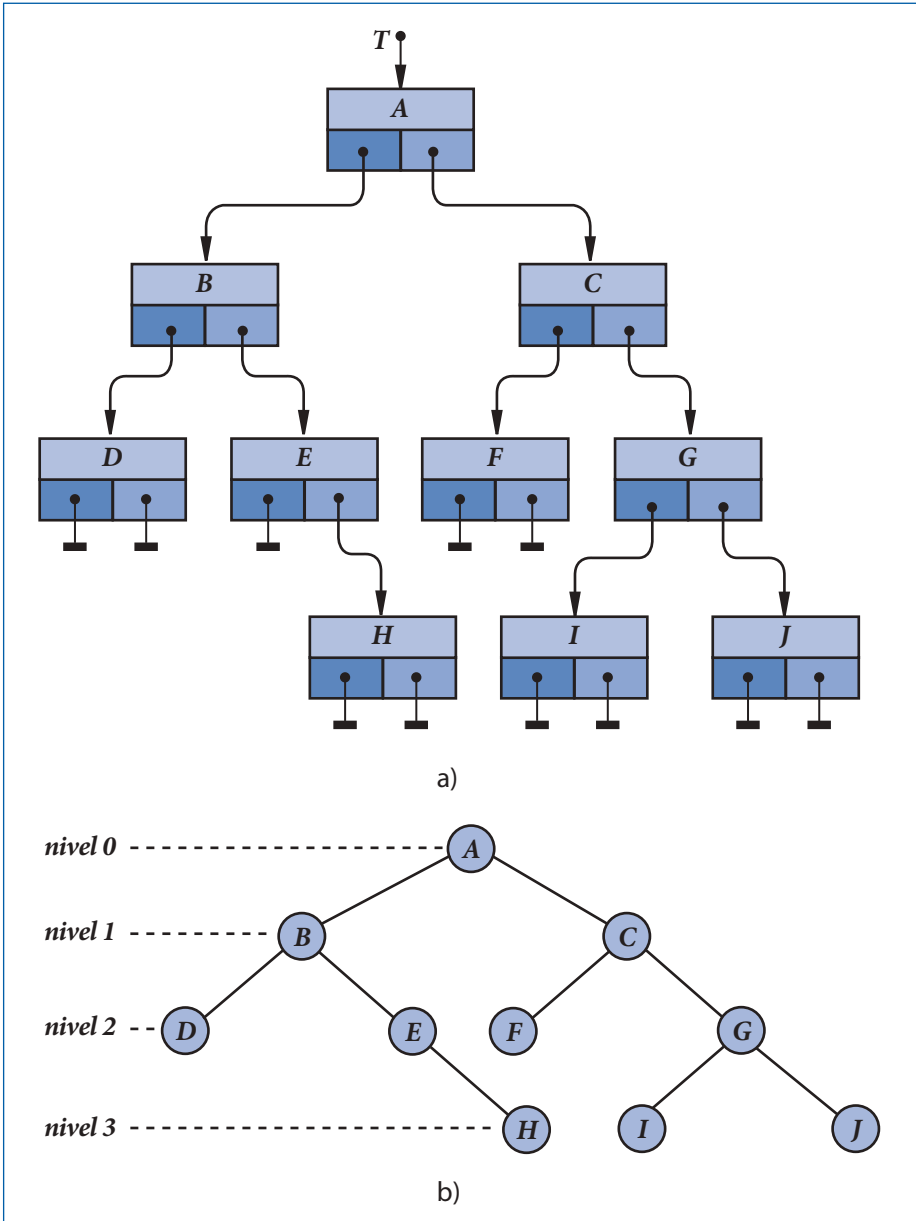


Fig. 2.11. Arborele binar:
a – reprezentarea detaliată; b – reprezentarea generalizată

Nodurile de pe nivelul $i + 1$, conectate la un nod de pe nivelul i , se numesc **descendenții** acestuia. În *figura 2.11* nodul B este descendentul stîng, iar nodul C este descendentul drept al nodului A ; nodul D este descendentul stîng, iar nodul E – descendentul drept al nodului B ș.a.m.d.

Dacă un nod x este descendentul altui nod y , îl numim pe acesta din urmă **părintele** nodului x . În *figura 2.11* nodul A este părintele nodurilor B și C ; nodul B este părintele nodurilor D și E ș.a.m.d.

Un nod la care nu este conectat niciun subarbore este un **nod terminal**. În caz contrar, nodul este **neterminal**. Prin **înălțimea** arborelui binar înțelegem numărul de nivel maxim asociat nodurilor terminale. Arborele din *figura 2.11* are înălțimea 3; nodurile D, H, F, I și J sînt noduri terminale; nodurile A, B, C, E și G sînt noduri neterminale.

Arborii binari pot fi construiți în memoria calculatorului cu ajutorul algoritmilor iterativi sau algoritmilor recursivi.

Algoritmul iterativ creează nodurile în ordinea apariției lor pe niveluri:

- se creează nodul-rădăcină;
- nodul-rădăcină se introduce într-o coadă;
- pentru fiecare nod extras din coadă se creează, dacă există, descendentul stîng și descendentul drept;
- nodurile nou-create se introduc în coadă;
- procesul de construire a arborelui se încheie cînd coada devine vidă.

Nodurile arborelui din *figura 2.11* vor fi create de algoritmul iterativ în ordinea: $A, B, C, D, E, F, G, H, I, J$.

Un algoritm similar poate fi utilizat pentru parcurgerea arborelui binar și afișarea nodurilor respective pe ecran:

- se creează o coadă care conține un singur element – nodul-rădăcină;
- fiecare nod extras din coadă este afișat pe ecran;
- descendenții nodului extras se introduc în coadă;
- procesul de afișare se încheie cînd coada devine vidă.

Exemplu:

```
Program P129;
{Crearea unui arbore binar - iterație }
type AdresaNod=^Nod;
     Nod=record
         Info : string;
         Stg, Dr : AdresaNod
     end;

     AdresaCelula=^Celula;
     Celula=record
         Info : AdresaNod;
         Urm : AdresaCelula
     end;

var T : AdresaNod;      {rădăcina }
    Prim,                {primul element din coadă }
    Ultim : AdresaCelula; { ultimul element din coadă }
```

```

procedure IntroduInCoada(Q : AdresaNod);
var R : AdresaCelula;
begin
    new(R);
    R^.Info:=Q;
    R^.Urm:=nil;
    if Prim=nil then begin Prim:=R; Ultim:=R end
        else begin Ultim^.Urm:=R; Ultim:=R end;
end; {IntroduInCoada}

procedure ExtrageDinCoada(var Q : AdresaNod);
var R : AdresaCelula;
begin
    if Prim=nil then writeln('Coadă este vidă')
        else begin
            R:=Prim;
            Q:=R^.Info;
            Prim:=Prim^.Urm;
            dispose(R);
        end;
end; { ExtrageDinCoadă }

procedure CreareArboreBinar;
var R, Q : AdresaNod;
    s : string;
begin
    T:=nil; {inițial arborele este vid }
    Prim:=nil; Ultim:=nil; {inițial coada este vidă }
    writeln('Dați rădăcina:'); readln(s);
    if s<>' ' then
        begin
            new(R); {crearea rădăcinii }
            R^.Info:=s;
            T:=R; {inițializarea adresei rădăcinii }
            IntroduInCoadă(T);
        end;
    while Prim<>nil do {cît coada nu e vidă }
        begin
            ExtrageDinCoadă(R);
            writeln('Dați descendenții nodului', R^.Info);
            write(' stîng: '); readln(s);
            if s=' ' then R^.Stg:=nil
                else
                    begin
                        new(Q); R^.Stg:=Q;
                        Q^.Info:=s;
                        IntroduInCoadă(Q);
                    end; { else }
        end;

```

```

write(' drept: '); readln(s);
  if s='' then R^.Dr:=nil
    else
      begin
        new(Q); R^.Dr:=Q;
        Q^.Info:=s;
        IntroduInCoadă(Q);
      end; { else }
    end; { while }
end; { CreareArboreBinar }
procedure AfisareArboreBinar;
var R : AdresaNod;
begin
  if T=nil then writeln('Arbore vid')
    else
      begin
        writeln('Arborele este format din:');
        Prim:=nil; Ultim:=nil;
        IntroduInCoadă(T);
        while Prim<>nil do
          begin
            ExtrageDinCoadă(R);
            writeln('Nodul ', R^.Info);
            write(' descendenți: ');
            if R^.Stg=nil then write('nil, ')
              else begin
                write(R^.Stg^.Info, ', ');
                IntroduInCoadă(R^.Stg);
              end;

            if R^.Dr=nil then writeln('nil')
              else begin
                writeln(R^.Dr^.Info);
                IntroduInCoadă(R^.Dr);
              end;
            end; { while }
          end; { else }
        readln;
      end; { AfisareArboreBinar }

begin
  CreareArboreBinar;
  AfisareArboreBinar;
end.

```

Informația utilă asociată fiecărui nod se citește de la tastatură. Absența descendentului se semnalează prin apăsarea tastei <ENTER> (programul citește de la tas-

tatură un șir vid de caractere). Menționăm că coada creată de programul P129 nu conține nodurile propriu-zise, ci adresele acestor noduri.

Algoritmul recursiv construiește arborii binari urmînd direct definiția respectivă:

- se creează nodul-rădăcină;
- se construiește subarborele stîng;
- se construiește subarborele drept.

Nodurile arborelui binar din *figura 2.11* vor fi create de algoritmul recursiv în ordinea: A, B, D, E, H, C, F, G, I, J.

Exemplu:

```
Program P130;
{Crearea unui arbore binar - recursie }
type Arbore=^Nod;
  Nod=record
    Info : string;
    Stg, Dr : Arbore
  end;

var T : Arbore;    {rădăcina }

function Arb : Arbore;
  {crearea arborelui binar }
var R : Arbore;
  s : string;
begin
  readln(s);
  if s='' then Arb:=nil
  else begin
    new(R);
    R^.Info:=s;
    write('Dați descendentul stîng');
    writeln(' al nodului ', s, ':');
    R^.Stg:=Arb;
    write('Dați descendentul drept');
    writeln(' al nodului ', s, ':');
    R^.Dr:=Arb;
    Arb:=R;
  end;
end; {Arb }

procedure AfisArb(T : Arbore; nivel : integer);
  {afisarea arborelui binar }
var i : integer;
begin
  if T<>nil then
```

```

begin
  AfisArb(T^.Stg, nivel+1);
  for i:=1 to nivel do write(' ');
  writeln(T^.Info);
  AfisArb(T^.Dr, nivel+1);
end;
end; {AfisareArb }

begin
  writeln('Dați rădăcina:');
  T:=Arb;
  AfisArb(T, 0);
  readln;
end.

```

Funcția `Arb` citește de la tastatură informația utilă asociată nodului în curs de creare. Dacă se citește un șir vid, nu se creează niciun nod și funcția returnează valoarea `nil`. În caz contrar, funcția creează un nod, înscrie șirul de caractere în câmpul `Info` și returnează adresa nodului. În momentul când trebuie completate câmpurile `Stg` (adresa subarborelui stîng) și `Dr` (adresa subarborelui drept), funcția se autoapelează, trecînd astfel la construcția subarborelui respectiv.

Procedura `AfisArb` afișează arborele binar pe ecran. Se afișează subarboarele stîng, rădăcina și apoi subarboarele drept. Nivelul fiecărui nod este redat prin inserarea numărului respectiv de spații.

Comparînd programele P129 și P130, se observă că prelucrarea structurilor recursive de date, și anume a arborilor binari, este mai naturală și mai eficientă în cazul utilizării unor algoritmi recursivi.

Arborii binari au numeroase aplicații, una dintre cele specifice fiind reprezentarea expresiilor în scopul prelucrării acestora în translaatoarele limbajelor de programare.

Întrebări și exerciții

- 1 Cum se definește un *arbore binar*? Explicați termenii: *rădăcină*, *subarboarele stîng*, *subarboarele drept*, *descendent*, *nivel*, *nod terminal*, *nod neterminal*, *înălțimea arborelui binar*.
- 2 Formulați algoritmi iterativi destinați creării și afișării arborilor binari.
- 3 Cum se construiește un arbore binar cu ajutorul algoritmului recursiv?
- 4 Elaborați un program care construiește arborele genealogic propriu pe parcursul a trei sau patru generații. Nodul-rădăcină conține numele, prenumele și anul nașterii, iar nodurile descendente conțin datele respective despre părinți.
- 5 Cum trebuie modificată procedura `AfisArb` din programul P130 pentru ca arborele binar să fie afișat în ordinea: subarboarele drept, nodul-rădăcină, subarboarele stîng?
- 6 Scrieți o funcție recursivă care returnează numărul nodurilor unui arbore binar. Transcrieți această funcție într-o formă nerecursivă.
- 7 Organizarea unui turneu „prin eliminare” este redată cu ajutorul unui arbore binar. Nodurile arborelui în studiu conțin următoarea informație:

- numele (**string**);
- prenumele (**string**);
- data nașterii (ziua, luna, anul);
- cetățenia (**string**).

Fiecărui jucător îi corespunde un nod terminal, iar fiecărui meci – un nod neterminal. În fiecare nod neterminal se înscriu datele despre câștigătorul meciului la care au participat cei doi jucători din nodurile descendente. Evident, rădăcina arborelui va conține datele despre câștigătorul turneului.

Scrieți un program care creează în memoria calculatorului și afișează pe ecran arborele unui turneu prin eliminare.

Indicație: Se pornește de la o listă a jucătorilor. Câștigătorii meciurilor din prima etapă se includ într-o altă listă. În continuare se formează lista câștigătorilor meciurilor din etapa a doua ș.a.m.d.

- 8 Cum trebuie modificată funcția `Arb` din programul P130 pentru ca arborele binar să se construiască în ordinea: *A, C, G, J, I, F, B, E, H, D*?
- 9 Funcția `Arb` din programul P130 construiește arborii binari în ordinea: nodul-rădăcină, subarborele stîng, subarborele drept. Scrieți o procedură nerecursivă care construiește arborii binari în aceeași ordine.

Indicație: Se utilizează o stivă elementele căreia sînt noduri. Inițial stiva va conține numai nodul-rădăcină. Pentru fiecare nod din vârful stivei se va construi subarborele stîng, iar apoi – subarborele drept. Nodurile nou-create se introduc în stivă. După construirea subarborelui drept, nodul respectiv este scos din stivă.

2.8. Parcurgerea arborilor binari

Operațiile care se pot efectua asupra arborilor binari se împart în două mari categorii:

- operații care modifică structura arborelui (inserarea sau eliminarea unui nod);
- operații care păstrează intactă structura arborelui (căutarea unei informații, tipărirea informațiilor asociate unui nod etc.).

Una din problemele care apar în mod frecvent la efectuarea acestor operații este necesitatea de a parcurge sau a traversa arborele binar.

Prin **parcurgerea unui arbore** se înțelege examinarea în mod sistematic a nodurilor sale astfel încît informația din fiecare nod să fie prelucrată o singură dată. Există trei modalități de parcurgere a arborilor binari: nodurile pot fi vizitate în preordine, inordine și postordine. Aceste trei metode sînt definite recursiv: dacă arborele este vid, atunci el este parcurs fără a se face nimic; astfel parcurgerea se face în trei etape.

Parcurgerea în preordine sau traversarea *RSD*:

- 1) se vizitează rădăcina;
- 2) se traversează subarborele stîng;
- 3) se traversează subarborele drept.

Parcurgerea în inordine sau traversarea *SRD*:

- 1) se traversează subarborele stîng;
- 2) se vizitează rădăcina;
- 3) se traversează subarborele drept.

Parcurgerea în postordine sau traversarea SDR:

- 1) se traversează subarborele stîng;
- 2) se traversează subarborele drept;
- 3) se vizitează rădăcina.

Notațiile RSD, SRD și SDR reprezintă ordinea în care vor fi vizitate rădăcina (R), subarborele stîng (S) și subarborele drept (D). Metodele de parcurgere a arborilor binari sînt ilustrate în figura 2.12.

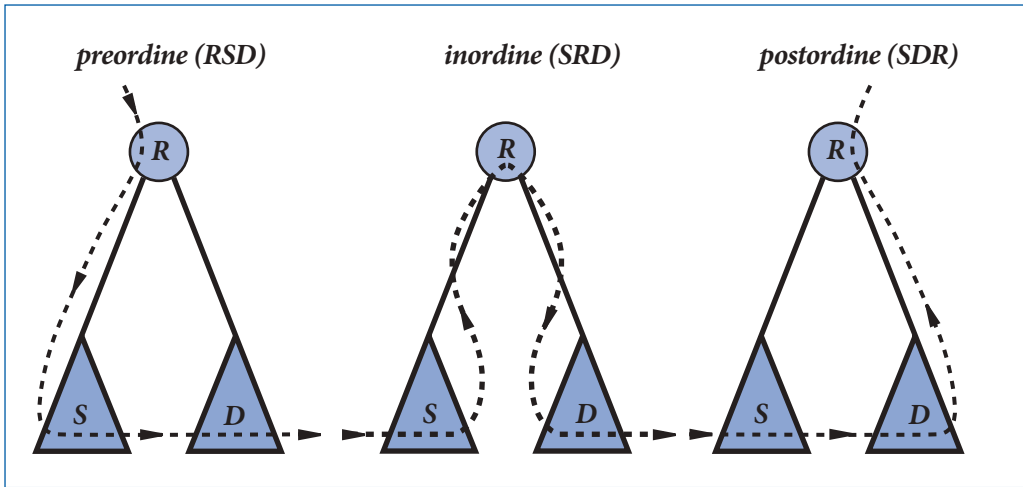


Fig. 2.12. Metodele de parcurgere a arborilor binari

Pentru arborele din figura 2.11 parcurgerea în preordine furnizează nodurile în ordinea:

A, B, D, E, H, C, F, G, I, J;

parcurgerea în inordine furnizează nodurile în ordinea:

D, B, E, H, A, F, C, I, G, J;

iar parcurgerea în postordine conduce la:

D, H, E, B, F, I, J, G, C, A.

Prezentăm mai jos un program PASCAL de parcurgere a unui arbore binar după toate cele trei metode.

```
Program P131;  
{Parcurgerea arborelui binar }  
type Arbore=^Nod;  
Nod=record  
    Info : string;  
    Stg, Dr : Arbore  
end;  
var T : Arbore; {rădăcina }
```

```

function Arb : Arbore;
  {crearea arborelui binar }
var R : Arbore;
      s : string;
begin
  readln(s);
  if s='' then Arb:=nil
    else begin
      new(R);
      R^.Info:=s;
      write('Dați descendentul stîng');
      writeln(' al nodului ', s, ':');
      R^.Stg:=Arb;
      write('Dați descendentul drept');
      writeln(' al nodului ', s, ':');
      R^.Dr:=Arb;
      Arb:=R;
    end;
end; {Arb }

procedure AfisArb(T : Arbore; nivel : integer);
  {afișarea arborelui binar }
var i : integer;
begin
  if T<>nil then
    begin
      AfisArb(T^.Stg, nivel+1);
      for i:=1 to nivel do write(' ');
      writeln(T^.Info);
      AfisArb(T^.Dr, nivel+1);
    end;
end; {AfisareArb }

procedure Preordine(T : Arbore);
  {traversare RSD }
begin
  if T<>nil then begin
      writeln(T^.Info);
      Preordine(T^.Stg);
      Preordine(T^.Dr)
    end;
end; {Preordine }

procedure Inordine(T : Arbore);
  {traversare SRD }
begin
  if T<>nil then begin

```



```

        Inordine(T^.Stg);
        writeln(T^.Info);
        Inordine(T^.Dr)
    end;
end; {Preordine }

procedure Postordine(T : Arbore);
{traversare SDR }
begin
    if T<>nil then begin
        Postordine(T^.Stg);
        Postordine(T^.Dr);
        writeln(T^.Info)
    end;
end; { Postordine }

begin
    writeln('Dați rădăcina:');
    T:=Arb;
    AfisArb(T, 0);
    readln;
    writeln('Parcurgere în preordine:');
    Preordine(T);
    readln;
    writeln('Parcurgere în inordine:');
    Inordine(T);
    readln;
    writeln('Parcurgere în postordine:');
    Postordine(T);
    readln;
end.

```

Menționăm că funcția `Arb` creează nodurile, parcurgând arborele binar în curs de construcție în preordine. Procedura `AfisArb` afișează nodurile, parcurgând arborele binar în inordine.

Întrebări și exerciții

- ❶ Ce operații pot fi efectuate asupra arborilor binari?
- ❷ Explicați metodele de parcurgere a arborilor binari. Dați exemple.
- ❸ Scrieți listele de noduri obținute în urma celor trei metode de parcurgere a arborelui binar din *figura 2.13*.
- ❹ Transcrieți procedurile `Preordine`, `Inordine` și `Postordine` din programul `P131` în formă nerecursivă.
- ❺ Scrieți un subprogram care returnează înălțimea arborelui binar.

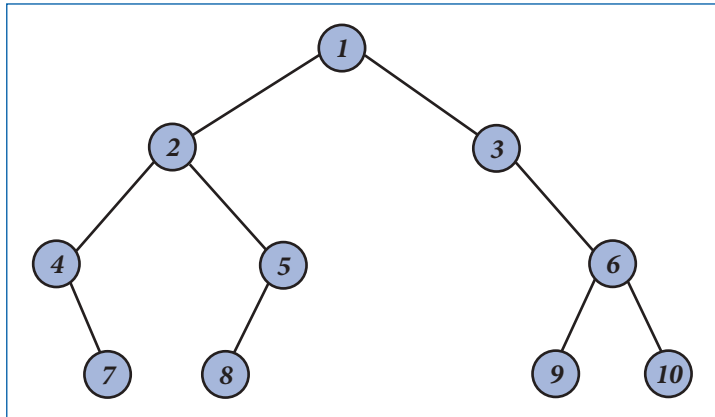


Fig. 2.13. Arborele binar

- 6 Elaborati un program care afiseaza pe ecran toate nodurile aflate pe un nivel dat intr-un arbore binar.
- 7 Elaborati o procedura recursiva care parcurge un arbore binar in ordinea:
 - a) RDS (radacina – subarboarele drept – subarboarele sting);
 - b) DRS (subarboarele drept – radacina – subarboarele sting);
 - c) DSR (subarboarele drept – subarboarele sting – radacina).
 Transcrieti procedura elaborata intr-o forma nerecursiva.
- 8 Scrieti un subprogram care afiseaza la ecran nivelul fiecarui nod dintr-un arbore binar.
- 9 Se da un arbore binar in care nodurile terminale reprezinta numere intregi, iar cele ne-terminale – operatiile binare +, -, *, mod, div. Arborele in studiu poate fi considerat ca o reprezentare a unei expresii aritmetice. Valoarea acestei expresii se calculeaza efectuind operatia din nodul-radacina asupra valorilor subexpresiilor reprezentate de subarboarele sting si subarboarele drept. Scrieti o functie care returneaza valoarea expresiilor aritmetice reprezentate prin arbori binari.
- 10 Se considera expresiile aritmetice formate din operanzi si operatorii binari +, -, *, /. Operanzii sint variabile numele carora este format dintr-o singura litera si constante alcătuite dintr-o cifra. Fiecarei expresii aritmetice i se poate asocia un arbore binar dupa cum urmeaza:
 - a) expresiei aritmetice formate dintr-un singur operand i se asociază un arbore binar format doar din nodul ce contine operandul respectiv;
 - b) expresiei aritmetice de forma $E_1 \circ E_2$, unde E_1 și E_2 sint expresii aritmetice, i se asociază un arbore binar care are in nodul-radacina operatorul \circ , ca subarboare sting arborele asociat expresiei E_1 , iar ca subarboare drept arborele asociat expresiei E_2 .
 Valoarea expresiei se calculeaza efectuind operatia din nodul-radacina asupra valorilor subexpresiilor reprezentate de subarboarele sting si subarboarele drept. Scrieti un program care:
 - a) construiesc arbori binari asociati expresiilor aritmetice citite de la tastatura;
 - b) evalueaza expresiile aritmetice reprezentate prin arborii binari.

Indicatie. Algoritmul va urma definitia recursiva a arborelui in studiu. Ca operator curent „ \circ ” se poate desemna orice operator +, - din expresia supusa prelucrării. Operatorii *, / pot fi desemnati ca operatori curenti numai cind expresia supusa prelucrării nu contine operatorii +, -.

2.9. Arbori de ordinul m

Se consideră variabile dinamice de tipul **record** care au în câmpul legăturilor indicatori de adresă. Ca și în cazul arborilor binari vom numi astfel de variabile **noduri**.

Arborele de ordinul m se definește recursiv după cum urmează:

a) un nod este un arbore de ordinul m ;

b) un nod ce conține cel mult m legături către alți arbori de ordinul m este un arbore de ordinul m .

Se consideră că în arbore există cel puțin un nod care subordonează exact m subarbori. Prin convenție, **arborele vid** nu conține niciun nod.

Arborii de ordinul 2 se numesc **arbori binari** și au fost studiați în paragrafele precedente. Arborii de ordinul 3, 4, 5 ș.a.m.d. se numesc **arbori multicăi** (în engleză *multivay tree*).

Pentru exemplificare, în *figura 2.14* este prezentat un arbore de ordinul 4. Evident, pentru arborii multicăi termenii rădăcină, subarbore, nivel, părinte, descendent, nod terminal, nod neterminal, înălțime au aceeași semnificație ca și pentru arborii binari. Terminologia utilizată în structurile de date în studiu include chiar cuvinte ca *fiu*, *tată*, *frați*, *unchi*, *veri*, *străbunic* etc. cu înțeles similar celui din vorbirea curentă pentru noduri aflate pe diverse niveluri. Într-un limbaj simplist, structurile de date în studiu exprimă relații de „ramificare” între noduri, asemănătoare configurației arborilor din natură, cu deosebirea că în informatică arborii „cresc” în jos.

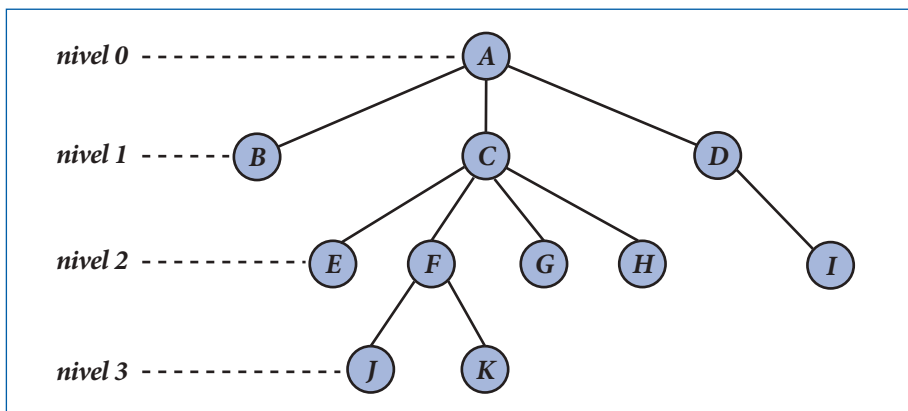


Fig. 2.14. Arborele de ordinul 4

Datele necesare pentru crearea și prelucrarea unui arbore binar de ordinul m pot fi definite prin declarații de forma:

```
type Arbore = ^Nod;
    Nod = record;
    Info : string;
    Dsc : array [1..m] of Arbore
    end;
var T : Arbore;
```

Adresele descendenților unui nod se memorează în componentele `Dsc[1]`, `Dsc[2]`, `...`, `Dsc[m]` ale tabloului `Dsc`. Adresa rădăcinii se reține în variabila de tip referință `T`.

Cele mai uzuale metode de parcurgere a arborilor de ordinul m sînt parcurgerea în lățime și parcurgerea în adîncime.

Parcurgerea în lățime presupune vizitarea nodurilor în ordinea apariției lor pe niveluri. De exemplu, pentru arborele din *figura 2.14* nodurile vor fi vizitate în ordinea: $A, B, C, D, E, F, G, H, I, J, K$.

În mod obișnuit, parcurgerea în lățime se realizează cu ajutorul unui algoritm iterativ care utilizează o structură auxiliară de date, și anume o coadă formată din adresele nodurilor care vor fi vizitate.

Parcurgerea în adîncime se definește recursiv: dacă arborele este vid, el este parcurs fără a se face nimic; altfel se vizitează întii rădăcina, apoi subarborii de la stînga la dreapta. Pentru arborele din *figura 2.14* parcurgerea în adîncime furnizează nodurile în ordinea: $A, B, C, E, F, J, K, G, H, D, I$. Parcurgerea în adîncime se realizează foarte simplu cu ajutorul unui algoritm recursiv.

Exemplu:

```
Program P133;
{Arbori de ordinul m }
const m=4;
type Arbore=^Nod;
     Nod=record
         Info : string;
         Dsc : array [1..m] of Arbore
     end;

     AdresaCelula=^Celula;
     Celula=record
         Info : Arbore;
         Urm : AdresaCelula
     end;

var T : Arbore;      {rădăcina }
    Prim,           {primul element din coadă }
    Ultim : AdresaCelula; {ultimul element din coadă }

procedure IntroduInCoadă(Q : Arbore);
var R : AdresaCelula;
begin
    new(R);
    R^.Info:=Q;
    R^.Urm:=nil;
    if Prim=nil then begin Prim:=R; Ultim:=R end
        else begin Ultim^.Urm:=R; Ultim:=R end;
end; {IntroduInCoadă }
```

```

procedure ExtrageDinCoada(var Q : Arbore);
var R : AdresaCelula;
begin
  if Prim=nil then writeln('Coadă este vidă')
    else begin
      Q:=R^.Info;
      Prim:=Prim^.Urm;
      dispose(R);
    end;
end; {ExtrageDinCoadă }

procedure CreareArbore(var T : Arbore);
var R, Q : Arbore;
    s : string;
    i : integer;
begin
  T:=nil; {inițial arborele este vid }
  Prim:=nil; Ultim:=nil; {inițial coada este vidă }
  writeln('Dați rădăcina: '); readln(s);
  if s<>' ' then
    begin
      new(R); {crearea rădăcinii }
      R^.Info:=s;
      T:=R; {inițializarea adresei rădăcinii }
      IntroduInCoadă(T);
    end;
  while Prim<>nil do {cât coada nu e vidă }
    begin
      ExtrageDinCoadă(R);
      for i:=1 to m do R^.Dsc [i]:=nil;
      writeln('Dați descendenții nodului',R^.Info);
      i:=1; readln(s);
      while (i<=m) and (s<>' ') do
        begin
          new(Q); R^.Dsc [i]:=Q; Q^.Info:=s;
          IntroduInCoadă(Q);
          i:=i+1; readln(s);
        end;
    end;
end; {CreareArbore }

procedure AfisareArbore(T : Arbore);
var R : Arbore;
    i : integer;
begin
  if T=nil then writeln('Arbore vid')

```

```

begin
  writeln('Arborele este format din:');
  Prim:=nil; Ultim:=nil;
  IntroduInCoadă(T);
  while Prim<>nil do
    begin
      ExtrageDinCoadă(R);
      writeln('Nodul ', R^.Info);
      write(' Descendenți: ');
      for i:=1 to m do
        if R^.Dsc [i]<>nil then
          begin
            write(R^.Dsc [i]^Info, ' ');
            IntroduInCoadă(R^.Dsc [i]);
          end; {then }
        writeln;
      end; {while }
    end; {else }
  readln;
end; {AfisareArbore }

procedure InLatime(T : Arbore);
var R : Arbore;
    i : integer;
begin
  if T<>nil then
    begin
      Prim:=nil; Ultim:=nil;
      IntroduInCoadă(T);
      while Prim<>nil do
        begin
          ExtrageDinCoadă(R);
          writeln(R^.Info);
          for i:=1 to m do
            if R^.Dsc [i]<>nil then IntroduInCoadă(R^.Dsc [i]);
          end; {while }
        end; {then }
      end; {InLatime }
    end;

procedure InAdincime(T : Arbore);
var i : integer;
begin
  if T<>nil then
    begin
      writeln(T^.Info);
      for i:=1 to m do InAdincime(T^.Dsc [i]);
    end;
  end; {InAdincime }

```

begin

```
CreareArbore(T);  
AfisareArbore(T);  
writeln('Parcurgerea arborelui în lățime:');  
InLatime(T);  
readln;  
writeln('Parcurgerea arborelui în adâncime:');  
InAdincime(T);  
readln;  
end.
```

Informația utilă asociată fiecărui nod se citește de la tastatură. Absența descendenților se semnalează prin apăsarea tastei <ENTER>. Menționăm că procedura *CreareArbore* creează nodurile parcurgând în lățime arborele în curs de construcție. Evident, procedura *AfisareArbore* vizitează nodurile în ordinea creării.

Operațiile frecvent efectuate asupra arborilor multicăi sînt: inserarea sau eliminarea unui nod, căutarea unei informații, prelucrarea informațiilor utile asociate nodurilor ș.a. De obicei, arborii multicăi sînt utilizați în cazul aplicațiilor care necesită prelucrarea unor mari cantități de date organizate ierarhic pe suporturile externe de informație. De exemplu, amintim modul de organizare a discurilor magnetice și optice în sistemele de operare MS-DOS, UNIX etc. Arborii în studiu sînt de asemenea utilizați în aplicațiile grafice pentru reprezentarea relațiilor dinamice dintre componentele imaginilor procesate.

Întrebări și exerciții

- 1 Dați exemple de arbori de ordinul 3, 5, 6.
- 2 Cum se definește un arbore de ordinul m ? Ce operații pot fi efectuate asupra arborilor în studiu?
- 3 Explicați metodele de parcurgere a arborilor multicăi. Dați exemple.
- 4 Scrieți un program recursiv care construiește în memoria calculatorului un arbore multicăi. Informația utilă asociată nodurilor se citește de la tastatură.
- 5 Scrieți o funcție care returnează:
 - a) numărul nodurilor unui arbore multicăi;
 - b) nivelul unui anumit nod din arbore;
 - c) înălțimea arborelui.
- 6 Transcrieți procedura *InAdincime* din programul P133 într-o formă nerecursivă.
- 7 Cum trebuie modificată procedura *InLatime* din programul P133 ca nodurile arborelui din *figura 6.18* să fie vizitate în ordinea: A, D, C, B, I, H, G, F, E, K, J?
- 8 Cum trebuie modificată procedura *InAdincime* din programul P133 pentru ca nodurile arborelui din *figura 2.14* să fie vizitate în ordinea: A, D, I, C, H, G, F, K, J, E, B?
- 9 Se dă un arbore multicăi, informațiile din noduri fiind șiruri de caractere. Să se afișeze pe ecran toate șirurile de caractere de lungime pară.

- ⑩ Organizarea datelor de pe discurile magnetice este redată cu ajutorul unui arbore multicăi. Nodurile terminale reprezintă fișierele, iar nodurile neterminale – directoarele. Informația utilă asociată fiecărui nod include:
- numele fișierului sau directorului (`string[8]`);
 - extensia (`string[3]`);
 - data și ora ultimei actualizări (respectiv ziua, luna, anul și ore, minute, secunde);
 - lungimea (`integer`);
 - atributele ('A', 'H', 'R', 'S').
- Elaborați un program care simulează operațiile de căutare, creare și ștergere a fișierelor și directoarelor.
- ⑪ În unele cazuri ordinul m al arborelui multicăi nu este cunoscut în momentul scrierii programului, fapt ce nu permite utilizarea structurilor de date de tipul `array[1..m] of Arbore`. Pentru a depăși acest inconvenient, tabloul respectiv poate fi înlocuit cu o listă uni- sau bidirecțională.
- Elaborați subprogramele necesare pentru crearea și prelucrarea arborilor multicăi de ordin arbitrar.

2.10. Tipul de date `pointer`

Acest paragraf se referă în întregime la implementarea Turbo PASCAL.

Mulțimea de valori ale tipului predefinit de date `pointer` (indicator) constă din adrese și valoarea specială `nil`. Însă, spre deosebire de tipurile de date referință adresele cărora identifică numai variabilele dinamice ce aparțin tipului de bază, valorile de tip `pointer` pot identifica variabile dinamice de orice tip. Evident, valoarea `nil` nu identifică nicio variabilă dinamică. Prin convenție, tipul de date `pointer` este **compatibil** cu orice tip de date referință.

Operațiile care se pot face cu valori de tipul de date `pointer` sînt = și <>. Valorile de acest tip nu pot fi citite de la tastatură și afișate pe ecran.

O variabilă de tip `pointer` se introduce printr-o declarație de forma:

```
var p : pointer;
```

Întrucît astfel de declarații nu conțin informații despre tipul de bază, tipul variabilei dinamice `p` este necunoscut. Prin urmare, variabilele de tip `pointer` nu pot fi dereperate, iar scrierea caracterului `^` după astfel de variabile constituie o eroare.

Programul ce urmează ilustrează utilizarea variabilelor de tip `pointer` pentru memorarea temporară a valorilor variabilelor de tip referință.

```
Program P134;
{ Tipul de date pointer }
var p : pointer;
    i, j : ^integer;
    x, y : ^real;
    r, s : ^string;
begin
    {p va identifica o variabilă dinamică de tipul integer }
    new(i); i^:=1;
```



```

p:=i;
new(i); i^:=2;
j:=p;
writeln('j^=', j^); { se afișează 1 }
{p va identifica o variabilă dinamică de tipul real }
new(x); x^:=1;
p:=x;
new(x); x^:=2;
y:=p; writeln('y^=', y^); {se afișează 1.0000000000E+00 }
{p va identifica o variabilă dinamică de tipul string }
new(r); r^:='AAA';
p:=r;
new(r); r^:='BBB';
s:=p;
writeln('s^=', s^); { se afișează AAA }
readln;
end.

```

Domeniul principal de utilizare a variabilelor de tip pointer este gestionarea memoriei interne a calculatorului. În Turbo PASCAL alocarea variabilelor dinamice se execută într-o zonă specială a memoriei interne numită **heap** (grămadă). Adresa de început a *heap*-ului, numită **adresa de bază**, este depusă în variabila predefinită de tip pointer `HeapOrg`. Variabila de tip pointer `HeapPtr` conține adresa primei locații libere, numită **vârful** *heap*-ului (fig. 2.15).

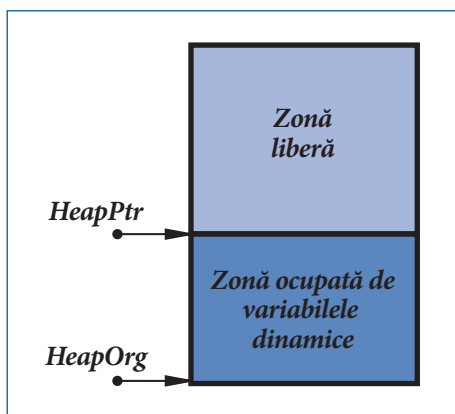


Fig. 2.15. Structura *heap*-ului

Variabilele dinamice sînt create și depuse în *heap* de procedura `new`. Ori de cîte ori în vîrful *heap*-ului se creează o variabilă dinamică conținutul variabilei `HeapPtr` este actualizat: valoarea curentă este incrementată cu dimensiunea spațiului de memorie necesar variabilei dinamice.

Memoria ocupată în *heap* de o variabilă dinamică se eliberează printr-un apel al procedurii `dispose`. Dimensiunea spațiului ce se eliberează depinde de tipul variabilei dinamice.

Ordinea de apelare a procedurii `dispose` nu coincide în general cu ordinea creării variabilelor dinamice de către procedura `new`. În consecință, în *heap* pot apărea goluri. Golurile apărute pot fi refolosite de procedura `new`, dacă variabila dinamică în curs de creare „încapă” în spațiul respectiv.

Eliberarea memoriei ocupate de o structură dinamică de date poate fi efectuată apelînd procedura `dispose` pentru fiecare componentă. Întrucît în program sînt cunoscute numai adresele componentelor privilegiate, de regulă baza și vîrfurile listei, rădăcina arborelui etc., căutarea celorlalte componente cade în sarcina programatorului. Mai mult decît afit, ordinea de apelare a procedurii `dispose` trebuie să asigure păstrarea legăturilor către componentele care încă nu au fost distruse. În caz contrar, componentele respective nu mai sînt referite de niciun indicator de adresă și devin inaccesibile. Prin urmare, utilizarea procedurii `dispose` pentru eliberarea memoriei ocupate de structuri complexe de date este greoaie și ineficientă. Acest inconvenient poate fi depășit cu ajutorul procedurilor predefinite `mark` și `release`.

Apelul procedurii `mark` are forma:

```
mark(p)
```

unde `p` este o variabilă de tip `pointer`. Procedura memorează adresa vîrfului din `HeapPtr` în variabila `p`.

Apelul procedurii `release` are forma:

```
release(p)
```

Această procedură reface adresa vîrfului în starea înregistrată anterior cu procedura `mark`: valoarea conținută în variabila de tip `pointer` `p` este depusă în indicatorul `HeapPtr`.

Zona de memorie destinată alocării variabilelor dinamice poate fi gestionată cu ajutorul algoritmului ce urmează:

- 1) se memorează adresa vîrfului cu procedura `mark`;
- 2) se creează variabilele dinamice cu procedura `new`;
- 3) se utilizează variabilele dinamice create;

4) cînd variabilele dinamice nu mai sînt necesare, spațiul ocupat din *heap* este eliberat cu procedura `release`.

Exemplu:

Se consideră următoarele declarații:

```
var i, j, k, m, n : ^integer;  
    p : pointer;
```

Să presupunem că sînt executate instrucțiunile:

```
new(i); i^:=1;  
new(j); j^:=2;  
mark(p);  
new(k); k^:=3;  
new(m); m^:=4;  
new(n); n^:=5;
```

Starea *heap*-ului este prezentată în *figura 2.16a*. Instrucțiunea `mark(p)` a memorat în variabila de tip pointer `p` valoarea actuală din `HeapPtr` înainte de crearea variabilei dinamice `k^`.

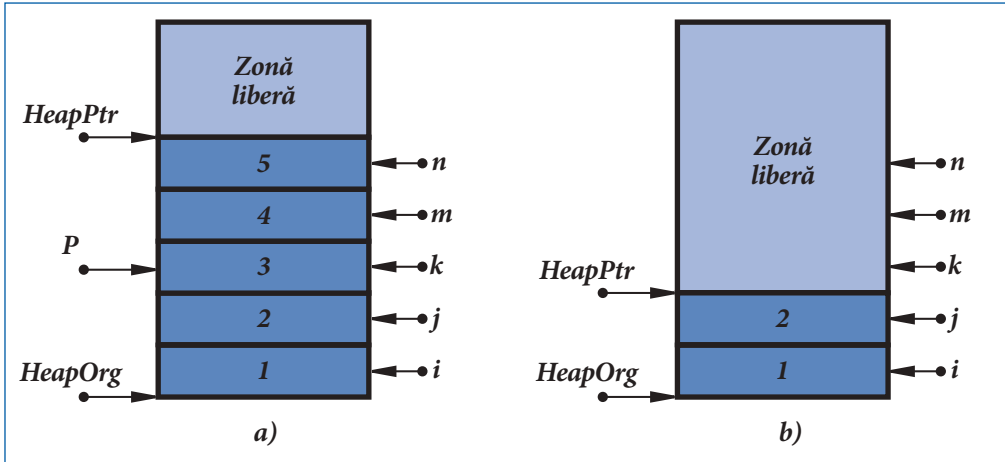


Fig. 2.16. Starea *heap*-ului pînă (a) și după executarea instrucțiunii `release(p)` (b)

Dacă acum se execută instrucțiunea

```
release(p)
```

memoria ocupată de variabilele dinamice create după apelul procedurii `mark`, și anume, `k^`, `m^` și `n^`, va fi eliberată (*fig. 2.16b*).

Deoarece variabila predefinită `HeapOrg` reține adresa de bază a *heap*-ului, tot spațiul destinat alocării variabilelor dinamice poate fi eliberat cu ajutorul instrucțiunii

```
release(HeapOrg)
```

Programul ce urmează ilustrează utilizarea procedurilor `mark` și `release`.

```
Program P135;
{Gestionarea memoriei interne }
type Lista=^Celula;
   Celula=record
       Info : string;
       Urm  : Lista
   end;
   Stiva=Lista;
end;
var L : Lista;
    S : Stiva;
    T : Arbore;
    p : pointer;

function Lst : Lista;
{crearea listei unidirecționale }
```

```

var R : Lista;
    s : string;
begin
write('Info='); readln(s);
if s='' then Lst:=nil
else
    begin
        new(R);
        R^.Info:=s;
        R^.Urm:=Lst;
        Lst:=R;
    end;
end; {Lst }

procedure AfisLst(L : Lista);
{afişarea listei }
begin
if L<>nil then
    begin
        writeln(L^.Info);
        AfisLst(L^.Urm);
    end;
end; {AfisLst }

procedure Stv(var S : Stiva);
{crearea unei stive }
var R : Stiva;
    st : string;
begin
S:=nil;
write('Info='); readln(st);
while st<>' ' do
    begin
        new(R);
        R^.Info:=st;
        R^.Urm:=S;
        S:=R;
        write('Info='); readln(st);
    end;
end; { Stv }

function Arb : Arbore;
{crearea arborelui binar }
var R : Arbore;
    s : string;

```

```

begin
  readln(s);
  if s='' then Arb:=nil
    else begin
      new(R);
      R^.Info:=s;
      write('Dați descendentul stîng');
      writeln(' al nodului ', s, ':');
      R^.Stg:=Arb;
      write('Dați descendentul drept');
      writeln(' al nodului ', s, ':');
      R^.Dr:=Arb;
      Arb:=R;
    end;
end; { Arb }

procedure AfisArb(T : Arbore; nivel : integer);
  { afișarea arborelui binar }
var i : integer;
begin
  if T<>nil then
    begin
      AfisArb(T^.Stg, nivel+1);
      for i:=1 to nivel do write(' ');
      writeln(T^.Info);
      AfisArb(T^.Dr, nivel+1);
    end;
end; {AfisArb }
begin
  writeln('Dați lista:');
  L:=Lst;
  writeln('Lista creată:');
  AfisLst(L);
  mark(p); { p reține adresa din HeapPtr }
  writeln('Dați rădăcina:');
  T:=Arb;
  writeln('Arborele creat:');
  AfisArb(T, 0);
  release(p);{eliberarea memoriei ocupate de arbore }
  writeln('Dați stiva:');
  Stv(S);
  writeln('Stiva creată');
  AfisLst(S);
  release(HeapOrg); {eliberarea memoriei ocupate de listă și
stivă }
  readln;
end.

```

Subliniem faptul că în implementările actuale procedurile `dispose` și `relea-` se nu atribuie valoarea `nil` indicatorilor de adresă variabilele dinamice ale cărora au fost distruse (fig. 2.16b). Întrucât memoria eliberată este refolosită, atribuirile efectuate asupra variabilelor distruse pot altera valorile variabilelor dinamice nou-create.

Întrebări și exerciții

- ❶ Care este mulțimea de valori ale tipului de date `pointer`? Ce operații pot fi efectuate cu aceste valori?
- ❷ Comentați următorul program:

```

Program P136;
  {Eroare }
  var i : ^integer;
      j, k : integer;
      p : pointer;
  begin
    new(i); i^:=1;
    p:=i;
    new(i); i^:=2;
    j:=i^; k:=p^;
    writeln('j+k=', j+k);
  end.

```

- ❸ Care este domeniul de utilizare a variabilelor de tip `pointer`?
- ❹ Este oare *heap*-ul o structură de date de tip stivă? Argumentați răspunsul.
- ❺ Lansați în execuție programele ce urmează. Explicați rezultatele afișate pe ecran.

```

Program P137;
  var i, j, k, m, n : ^integer;
      p : pointer;
  begin
    {crearea variabilelor i^, j^, k^ }
    new(i); new(j); new(k);
    i^:=1; j^:=2; k^:=3;
    p:=j; {p reține adresa din j }
    {distrugerea variabilei j^ și crearea variabilei m^}
    dispose(j); new(m); m^:=4;
    j:=p; {refacerea adresei din j }
    writeln('i^=', i^, ' j^=', j^, ' k^=', k^);
    {distrugerea variabilei m^ și crearea variabilei n^ }
    dispose(m); new(n); n^:=5;
    writeln('i^=', i^, ' j^=', j^, ' k^=', k^);
    readln;
  end.

```

```

Program P138;
var i, j, k, m : ^integer;
begin

  {crearea variabilelor i^, j^ }
  new(i); new(j);
  i^:=1; j^:=2;

  {eliberarea memoriei heap-ului }
  release(HeapOrg);

  {crearea variabilelor k^ și m^ }
  new(k); new(m);

  k^:=1; m^:=2;
  writeln('k^=', k^, ' m^=', m^);
  i^:=3; j^:=4;
  writeln('k^=', k^, ' m^=', m^);
  readln;
end.

```

6 Scrieți o procedură care eliberează memoria ocupată de:

- a) o listă unidirecțională;
- b) un arbore binar;
- c) un arbore multicăi.

Memoria trebuie eliberată apelând procedura `dispose` pentru fiecare componentă a structurii dinamice de date.

7 Scrieți un program în care se creează mai întâi o coadă, iar apoi un arbore multicăi. Spațiul de memorie eliberat după distrugerea cozii trebuie refolosit pentru alocarea arborelui.

METODE DE ELABORARE A PRODUSELOR PROGRAM

3.1. Programarea modulară

Programarea modulară urmărește reducerea complexității programelor mari prin descompunerea acestora în module.

Modulul este un produs program format din descrieri de date și subprograme destinate prelucrării acestora. Modulele pot fi scrise independent și compilate separat înainte de a fi încorporate în programul în curs de elaborare. Menționăm că pentru program se mai utilizează și denumirea de **modul principal**.

Limbajul-standard nu prevede mijloace pentru programarea modulară. Se consideră că programele PASCAL sînt entități monolit care trebuie compilate împreună cu subprogramele pe care, eventual, le conțin. Acest lucru devine incomod în cazul programelor mari care pot include zeci și chiar sute de subprograme.

În versiunea Turbo PASCAL modulele sînt implementate prin **unități de program**. Forma generală a unei **unități de program** este:

```
unit <Nume>;
interface
[uses <Nume> { ,<Nume>} ; ]
[<Constante>]
[<Tipuri>]
[<Variabile>]
[ {<Antet funcție>; | <Antet procedură>; } ]

implementation
[uses <Nume> { ,<Nume>} ; ]
[<Etichete>]
[<Constante>]
[<Tipuri>]
[<Variabile>]
[<Subprograme>]
[ {function <Identificator>;
<Corp>; |
procedure <Identificator>;
<Corp>; } ]
[begin
<Instrucțiune> { ; <Instrucțiune } ]
end.
```


În esență, o unitate de program constă din trei secțiuni: de interfață, de implementare și de inițializare.

Secțiunea de interfață începe cu cuvântul-cheie **interface**. Aici se declară constantele, tipurile, variabilele și subprogramele exportate de unitate. Aceste elemente pot fi referite de orice modul care utilizează direct sau prin tranzitivitate unitatea respectivă. Menționăm că în secțiunea de interfață apar doar antetele funcțiilor și procedurilor exportate. Dacă unitatea actuală utilizează alte unități, numele acestora sînt specificate în clauza **uses**.

Secțiunea de implementare începe cu cuvântul-cheie **implementation**. Această secțiune conține declarații locale de etichete, constante, tipuri, variabile și subprograme. Elementele definite aici sînt „ascunse” și nu pot fi referite de modulele care utilizează unitatea actuală. După declarațiile locale urmează corpul procedurilor și funcțiilor, ale căror antete au fost definite în secțiunea de interfață. Fiecare subprogram specificat în interfață trebuie să aibă un corp. După cuvântul-cheie **function** sau **procedure** se scrie doar numele subprogramului. Menționăm că nu este necesară descrierea listei de parametri și a valorii returnate.

Secțiunea de inițializare începe, dacă există, cu cuvântul-cheie **begin**. Secțiunea constă dintr-o secvență de instrucțiuni și servește pentru atribuirea valorilor inițiale variabilelor definite în secțiunea de interfață. Dacă un program utilizează mai multe unități, execuția programului este precedată de execuția secțiunilor de inițializare în ordinea în care aceste unități apar în clauza **uses** din program.

Exemplu:

```
Unit U1;
  {Prelucrarea vectorilor }
interface

  const nmax=100;
  type Vector=array [1..nmax] of real;
  var n : 1..nmax;
  function sum(V : Vector) : real;
  function min(V : Vector) : real;
  function max(V : Vector) : real;
  procedure Citire(var V : Vector);
  procedure Afisare(V : Vector);

implementation

  var i : 1..nmax;
      s : real;

  function sum;
  begin
    s:=0;
    for i:=1 to n do s:=s+V [i];
    sum:=s;
  end; {sum }
```

```

function min;
begin
    s:=V [1];
    for i:=2 to n do
        if s>V [i] then s:=V [i];
    min:=s;
end; {min }

function max;
begin
    s:=V [1];
    for i:=2 to n do
        if s<V [i] then s:=V [i];
    max:=s;
end; {max }

procedure Citire;
begin
    for i:=1 to n do readln(V [i]);
end; {Citire }

procedure Afisare;
begin
    for i:=1 to n do writeln(V [i]);
end; {Afisare }

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

```

Unitatea U1 exportă constanta nmax, tipul Vector, variabila n, funcțiile sum, min, max, procedurile Citire și Afisare. Valoarea inițială a variabilei n este citită de la tastatură. Elementele în studiu pot fi referite în orice program ce conține clauza uses U1.

Exemplu:

```

Program P139;
    {Utilizarea unității U1 }
uses U1;
var A : Vector;
begin
    writeln('Dați un vector:');
    Citire(A);
    writeln('Ați introdus:');
    Afisare(A);
    writeln('sum=', sum(A));

```

```
writeln('min=', min(A));
writeln('max=', max(A));
readln;
end.
```

Domeniile de vizibilitate ale declarațiilor din unitățile de program se stabilesc conform regulilor ce urmează.

1. Declarațiile din secțiunea de implementare sînt vizibile numai în unitatea actuală.

2. Declarațiile din secțiunea de interfață sînt vizibile în:

- unitatea actuală;
- modulele care utilizează direct unitatea actuală;
- modulele care utilizează unitatea actuală prin tranzitivitate.

Referirea oricărui identificator *id* declarat într-o unitate *v* utilizată prin tranzitivitate se face prin *v.id*.

3. Dacă unul și același identificator este declarat în mai multe module, este luată în considerare declarația cea mai recentă.

Exemplu:

```
Program P140;
uses U2;
var x : integer;
begin
  x:=4;
  writeln('Programul P140:');
  writeln('n=', U3.n);
  writeln('m=', m);
  writeln('x=', x);
  readln;
end.
```

```
Unit U2;
interface
  uses U3;
  var m : integer;
      x : real;
implementation
begin
  writeln('Unitatea U2:');
  m:=2;
  writeln('  m=', m);
  x:=3.0;
  writeln('  x=', x);
end.
```

```

Unit U3;
interface
  var n : integer;
implementation
begin
  writeln('Unitatea U3:');
  n:=1;
  writeln('n=', n);
end.

```

În programul P140 unitatea U2 este utilizată direct, iar unitatea U3 prin tranzitivitate. Variabila *n* din modulul U3 este referită prin U3.n. Identificatorul *x* apare în declarațiile **var** *x*: *real* din unitatea U2 și **var** *x*: *integer* din programul P140. Compilatorul ia în considerare ultima declarație.

Unitățile de program se clasifică în unitățile-standard, livrate odată cu compilatorul *Turbo PASCAL*, și unitățile scrise de utilizator. În continuare prezentăm o caracteristică succintă a unităților-standard frecvent utilizate.

System – conține toate subprogramele predefinite din *Turbo PASCAL*. Unitatea în studiu se încorporează automat în toate programele, fără a fi necesară clauza **uses**.

Crt – permite utilizarea funcțiilor și procedurilor referitoare la lucrul cu ecranul în mod *text*, precum și cu tastatura și difuzorul. Accesibilitatea subprogramelor se realizează prin clauza **uses crt**.

Graph – implementează subprogramele destinate unor prelucrări grafice: definiții de ferestre și pagini, definiții de culori și palete, desenarea arcurilor, cercurilor, poligoanelor și a altor figuri, salvarea imaginilor etc. Serviciile unității de program pot fi accesate prin clauza **uses graph**.

Printer – asigură redirectarea operațiilor de scriere în fișierul *text* cu numele *lst* la imprimantă. Utilizând unitatea în studiu, programatorul nu mai trebuie să declare, să deschidă și să închidă acest fișier. Serviciile unității **Printer** devin accesibile unui program sau unei unități de program prin specificarea clauzei **uses printer**.

Destinația și modul de utilizare a constantelor, tipurilor de date, variabilelor, funcțiilor și procedurilor din unitățile-standard este inclusă în ghidurile de utilizare și sistemele de asistență *Turbo PASCAL's Online Help*.

Elaborând propriile unități de program, orice utilizator își poate crea biblioteci de subprograme ce descriu algoritmi din diverse domenii: rezolvarea ecuațiilor, calcule statistice, procesarea textelor, crearea și prelucrarea structurilor dinamice de date etc. Divizarea unui program mare în module ușurează activitatea de elaborare a produselor program în echipă. În astfel de cazuri fiecare programator scrie, testează și documentează câteva module relativ simple, ceea ce contribuie la îmbunătățirea produsului program rezultat.

Întrebări și exerciții

- ❶ Care sînt avantajele programării modulare? Prevede oare limbajul-standard mijloace pentru programarea modulară?
- ❷ Care este forma-standard a unei unități de program? Precizați structura și destinația secțiunilor de interfață, de implementare și de inițializare.

- 3 Cum se determină domeniile de vizibilitate ale declarațiilor din unitățile de program?
- 4 Precizați ce va afișa pe ecran programul ce urmează.

```
Program P141;  
uses U4;  
var s : string;  
begin  
  s:='BBB';  
  writeln('U5.k=', U5.k);  
  writeln('U5.m=', U5.m);  
  writeln('U5.s=', U5.s);  
  writeln('U4.m=', U4.m);  
  writeln('U4.s=', U4.s);  
  writeln('m=', m);  
  writeln('s=', s);  
  readln;  
end.
```

```
Unit U4;  
interface  
uses U5;  
var m : real;  
  s : char;  
implementation  
begin  
  m:=4.0;  
  s:='A';  
end.  
Unit U5;  
interface  
var k, m : integer;  
  s : real;  
implementation  
begin  
  k:=1;  
  m:=2;  
  s:=3.0;  
end.
```

- 5 Comentați programul:

```
Program P142;  
  {Eroare }  
uses U6;  
begin  
  writeln('k=', k);  
  writeln('m=', m);  
  readln;  
end.
```

```

Unit U6;
interface
  var k : integer;
implementation
  var m : integer;
begin
  k:=1;
  m:=2;
end.

```

- 6 Completați unitatea de program U1 din paragraful în studiu cu un subprogram care:
- returnează media aritmetică a componentelor unui vector;
 - aranjează componentele în ordine crescătoare;
 - returnează produsul componentelor unui vector;
 - returnează numărul componentelor pozitive;
 - aranjează componentele în ordine descrescătoare.
- 7 Scrieți o unitate de program care oferă descrieri de date și subprograme pentru prelucrarea matricelor.
- 8 Numerele întregi n , $n \leq 10^{254}$, pot fi reprezentate în calculator prin șiruri formate din caracterele '+', '-', '0', '1', '2', ..., '9'. Elaborați o unitate de program care conține funcțiile și procedurile necesare pentru efectuarea următoarelor operații:
- citirea numerelor de la tastatură;
 - afișarea numerelor pe ecran;
 - +, -, *, mod, div;
 - calcularea factorialului;
 - citirea și scrierea numerelor în fișiere secvențiale.
- 9 Șirurile de caractere de lungime n , $n \leq 500$, pot fi reprezentate în programele Turbo PASCAL prin variabile de tipul:

```

type lungime = 0..500;
  SirDeCaractere = record
    n: lungime;
    s: array [1..500] of char
  end;

```

Elaborați o unitate de program care conține funcțiile și procedurile necesare pentru efectuarea următoarelor operații:

- citirea șirurilor de la tastatură;
 - afișarea șirurilor pe ecran;
 - concatenarea șirurilor;
 - compararea lexicografică;
 - calcularea lungimii unui șir.
- 10 Elaborați o unitate de program pentru prelucrarea:
- listelor unidirecționale;
 - cozilor;
 - stivelor;

- d) arborilor binari;
- e) arborilor multicăi.

Utilizați în acest scop declarațiile de tipuri, funcții și proceduri din capitolul 2.

- ⑩ Găsiți în sistemul de asistență *Turbo PASCAL's Online Help* descrierea unităților-standard instalate pe calculatorul dvs. Afișați pe ecran textul fiecărei unități, determinați destinația și modul de utilizare a funcțiilor și procedurilor respective.

3.2. Testarea și depanarea programelor

Un program este **corect** dacă:

- a) după lansarea în execuție procesul de calcul se termină;
- b) rezultatele obținute reprezintă o soluție a problemei pentru rezolvarea căreia a fost scris programul.

În caz contrar programul conține **erori**.

Asigurarea corectitudinii unui program presupune execuția sa pentru fiecare combinație posibilă de valori ale datelor de intrare. În majoritatea cazurilor acest lucru este imposibil, deoarece domeniul de valori al datelor de intrare este, practic, infinit, iar soluțiile respective sînt necunoscute.

Testarea este o etapă în elaborarea programelor ce are drept scop eliminarea erorilor. Ea se realizează executînd programul cu anumite seturi de date de intrare numite **date de testare** sau, mai simplu, **teste**. În funcție de modul de selecție a datelor de testare, deosebim:

- testarea funcțională sau metoda cutiei negre;
- testarea structurală sau metoda cutiei transparente.

Amintim că termenul *cutie neagră* este folosit pentru un sistem, structura internă a căruia este necunoscută.

În cazul **testării funcționale** datele de testare sînt astfel concepute, încît să se asigure că fiecare funcție a programului este pe deplin realizată. Programul este văzut ca o cutie neagră, a cărei funcționare este determinată prin introducerea unor date și analiza rezultatelor obținute. Selectarea datelor de intrare depinde, în mare măsură, de îndemînarea și experiența celui care efectuează testarea. În mod obișnuit, se selectează **valori tipice** și **valori netipice** din domeniul datelor de intrare.

Exemplu. Se consideră programul P143. Textul programului nu este deocamdată prezentat pentru a sublinia faptul că în metoda testării funcționale el nu este necesar. Programul realizează următoarele funcții:

- citește de la tastatură un șir de numere reale;
- afișează pe ecran media aritmetică a numerelor pozitive din șir.

Evident, domeniul datelor de intrare este format din șiruri de numere reale.

Datele de testare vor include:

- a) valorile netipice:
 - șir vid;
 - șir ce nu conține numere pozitive;
 - șir ce conține un singur număr pozitiv;

b) valorile tipice:

- șir cu două numere pozitive;
- șir cu trei sau mai multe numere pozitive.

În **testarea structurală** testele sînt elaborate examinînd structura programului: declarațiile de date, proceduri și funcții, instrucțiunile simple, instrucțiunile structurale etc. Datele de testare vor asigura:

- a) execuția fiecărei instrucțiuni simple (atribuiri, apeluri de proceduri, salturi **goto**);
- b) execuția fiecărui ciclu **for** de zero, unu și de mai multe ori;
- c) execuția fiecărei instrucțiuni **if**, **repeat**, **while** pentru valorile true și false ale expresiilor booleene de control;
- d) execuția fiecărui caz din componența instrucțiunilor **case**.

Exemplu. Prezentăm textul programului care calculează media aritmetică a numerelor pozitive dintr-un șir:

```
Program P143;
  { Media numerelor pozitive dintr-un șir }
var n, k : integer;
    x, s : real;
begin
  n:=0;
  k:=0;
  s:=0;
  writeln('Dați un șir de numere reale:');
  while not eof do
    begin
      readln(x);
      n:=n+1;
      if x>0 then
        begin
          k:=k+1;
          s:=s+x;
        end;
    end; { while }
  if n=0 then writeln('șir vid')
    else if k=0 then writeln('Șirul nu conține numere
                             pozitive')
      else writeln('Media=', s/k);
  readln;
end.
```

Testul trebuie să asigure execuția instrucțiunilor **while** și **if** pentru valorile true și false ale expresiilor booleene **not eof**, $x > 0$, $n = 0$ și $k = 0$. Prin urmare, datele de testare vor include:

- un șir vid (**not eof**=false, $n=0$);
- un șir nevid (**not eof**=true, $n \neq 0$);
- un șir ce conține cel puțin un număr pozitiv ($x > 0$, $k \neq 0$);
- un șir nevid ce nu conține numere pozitive ($x \leq 0$, $k = 0$).

Metoda cutiei transparente poate fi utilizată independent sau împreună cu metoda cutiei negre pentru a îmbunătăți un test deja obținut. De exemplu, în cazul programului P143 șirul ce conține cel puțin un număr pozitiv poate fi înlocuit cu trei șiruri ce conțin, respectiv unul, două, trei sau mai multe numere pozitive. Aceste date vor asigura execuția instrucțiunilor $k:=k+1$, $s:=s+x$ și calcularea expresiei s/k pentru valorile tipice și valorile netipice ale variabilelor k și s .

Dificultățile în aplicarea testării structurale sînt legate de prezența instrucțiunilor de decizie (**if, case**), a celor iterative (**for, while, repeat**) sau a celei de transfer (**goto**). Acestea determină apariția unui număr foarte mare de combinații, în care instrucțiunile de atribuire și apelurile de proceduri pot fi executate.

Depanarea programului constă în localizarea zonelor din program care au condus la apariția unei erori, identificarea cauzelor erorii și corectarea acesteia. Depanarea poate fi făcută static (după executarea programului) și dinamic (în timpul executării).

În metoda **depanării statice** cauzele erorii se stabilesc analizînd rezultatele derulării programului și mesajele sistemului de operare. Pentru a facilita procesul de depanare, în program temporar se includ instrucțiuni care afișează pe ecran valorile intermediare ale variabilelor-cheie.

În metoda **depanării dinamice** localizarea erorilor se face urmărind executarea programului la nivel de instrucțiuni. Implementările actuale ale limbajului permit efectuarea următoarelor operații de depanare dinamică:

- execuția pas cu pas a programului;
- observarea valorilor unor expresii specificate;
- crearea și eliminarea unor puncte de suspendare a executării;
- modificarea valorilor unor variabile;
- trasarea apelurilor de funcții și proceduri;
- tratarea erorilor de intrare-ieșire, a erorilor de depășire etc.

Descrierea detaliată a operațiilor în studiu este inclusă în sistemul de asistență *Turbo PASCAL's Online Help*.

Eficiența depanării depinde de modul în care este scris și testat programul, calitatea mesajelor de eroare generate de calculator și tipul erorii. De regulă, un test care semnalează prezența unei erori este urmat de alte texte organizate în așa fel, încît să izoleze eroarea și să furnizeze informații pentru corectarea ei.

S-a constatat că testarea și depanarea ocupă mai mult de jumătate din perioada de timp necesară realizării unui produs program. Complexitatea acestor procese poate fi redusă prin divizarea programelor mari în subprograme sau module și aplicarea metodelor programării structurate.

Subliniem faptul că testarea programelor este un mijloc eficient de a **depista erorile**, însă nu și un mijloc de a demonstra absența lor. Cu toate că testarea nu demonstrează corectitudinea programului, ea este deocamdată singura metodă practică de certificare a produselor program. În prezent se elaborează metode de verificare baze pe demonstrarea formală a corectitudinii programului, însă rezultatele cunoscute în această direcție nu sînt aplicabile programelor complexe.

Întrebări și exerciții

- ❶ Când un program PASCAL este corect? Cum poate fi asigurată corectitudinea unui program?
- ❷ Cum se selectează datele de intrare în metoda testării funcționale?
- ❸ Elaborati un test funcțional pentru programul P124 din paragraful 2.4. Programul realizează următoarele funcții:
 - creează o listă unidirecțională;
 - afișează lista pe ecran;
 - include un anumit element în listă;
 - exclude din listă elementul specificat de utilizator.
- ❹ Precizați funcțiile realizate de programele ce urmează și elaborati testele funcționale:
 - a) P117 și P120 din paragraful 2.1;
 - b) P122 și P123 din paragraful 2.2;
 - c) P127 din paragraful 2.5 și P128 din paragraful 2.6.
- ❺ Cum se selectează datele de intrare în metoda testării structurale?
- ❻ Elaborati teste structurale pentru programele ce urmează:
 - a) P117 și P120 din paragraful 2.1;
 - b) P122 și P123 din paragraful 2.2;
 - c) P127 din paragraful 2.5.
- ❼ Care este diferența dintre *depanarea statică* și *depanarea dinamică*?
- ❽ Găsiți în sistemul de asistență *Turbo PASCAL's Online Help* descrierea operațiilor de depanare dinamică. Efectuați aceste operații pentru programele elaborate de dvs.

3.3. Elemente de programare structurată

Încă din primii ani de activitate în domeniul prelucrărilor de date s-a constatat că testarea, depanarea și modificarea programelor necesită un mare volum de muncă. Mai mult decât atât, programele complexe ce conțin sute și mii de instrucțiuni devin greu accesibile chiar și pentru autorii lor.

Programarea structurată reprezintă un stil, o manieră de concepere a programelor potrivit unor reguli bine stabilite, bazate pe teorema de structură. Conform **teoremei de structură**, orice algoritm poate fi reprezentat ca o combinație a trei structuri de control:

- secvența (succesiune de două sau mai multe atribute și/sau apeluri de proceduri);

- decizia (**if... then... sau if... then... else...**);

- ciclul cu test inițial (**while... do...**).

Programarea structurată admite și utilizarea altor structuri de control, cum sînt:

- selecția (**case... of...**);

- ciclul cu test final (**repeat... until...**);

- ciclul cu contor (**for... do...**).

Regulile de bază ale programării structurate sînt:

1. Structura oricărui program sau subprogram va fi concepută ca o combinație a structurilor de control admise: secvența, decizia, selecția, ciclul.

2. Structura datelor utilizate în program trebuie să corespundă specificului problemelor rezolvate.

3. Lungimea maximă a unei funcții sau proceduri este de 50–100 de linii. Folosirea variabilelor globale nu este încurajată.

4. Identificatorii folosiți pentru constante, tipuri, variabile, funcții, proceduri și unități de program trebuie să fie cît mai sugestivi.

5. Claritatea textului trebuie asigurată prin inserarea comentariilor și alinierea textului în conformitate cu structura logică și sintactică a instrucțiunilor.

6. Operațiile de intrare-ieșire vor fi localizate în subprograme separate. Corectitudinea datelor de intrare se verifică imediat după citirea lor.

7. Încuibarea instrucțiunilor **if** mai mult de trei ori trebuie evitată prin folosirea instrucțiunilor **case**.

Programele obținute conform regulilor în studiu sînt testabile, clare, ordonate, fără salturi și reveniri. Menționăm că, conform teoremei de structură, orice program poate fi scris fără a utiliza instrucțiunea **goto**. Totuși unii autori admit utilizarea acestei instrucțiuni cu condiția ca ea să fie folosită la minimum, iar salturile să fie efectuate numai în jos.

Întrebări și exerciții

- 1 Care este justificarea teoretică a programării structurate?
- 2 Precizați structurile de control necesare și suficiente pentru reprezentarea oricărui algoritm.
- 3 Formulați regulile de bază ale programării structurate.
- 4 Care sînt avantajele programării structurate?
- 5 Corespund oare programele P124, P130 și P135 din capitolul 2 regulilor de bază ale programării structurate?
- 6 Programul ce urmează afișează pe ecran toate reprezentările posibile ale numărului natural n ca sumă de numere naturale consecutive.

```
Program P144;  
var a,i,l,s,n : integer;  
b : boolean;  
begin  
write('n='); readln(n);  
b:=true;  
for i:=1 to ((n+1) div 2) do  
begin  
a:=i;  
s:=0;  
while (s<n) do  
begin  
s:=s+a;  
a:=a+1;  
end;  
end;
```

```

if s=n then
begin
  b:=false;
  write(n, '=', i);
  for l:=i+1 to (a-1) do write('+',l);
  writeln;
end;
end;
if b then writeln('Reprezentări nu există');
  readln;
end.

```

De exemplu, pentru $n = 15$ obținem:

15=1+2+3+4+5;

15=4+5+6;

15=7+8.

Reprezentările în studiu se calculează prin metoda trierii, examinându-se șirurile:

1, 2, 3, ..., k ;

2, 3, ..., k ;

3, ..., k

ș.a.m.d., unde $k = (n + 1) \text{ div } 2$. Termenii 1, 2, 3, ..., k se calculează cu ajutorul relației recurente $a = a + 1$. Aliniați textul programului în conformitate cu structura logică și sintaxa fiecărei instrucțiuni.

ANALIZA ALGORITMILOR

4.1. Complexitatea algoritmilor

Valoarea practică a programelor PASCAL depinde în mod decisiv de complexitatea algoritmilor ce stau la baza lor. Amintim că algoritmul reprezintă o succesiune finită de operații (instrucțiuni, comenzi) cunoscute, care, fiind executate într-o ordine bine stabilită, furnizează soluția unei probleme.

E cunoscut faptul că unul și același algoritm poate fi descris prin diverse metode: scheme logice, formule matematice, texte scrise într-un limbaj de comunicare între oameni, cu ajutorul limbajelor de programare. Evident, și în acest manual algoritmi pe care îi vom studia vor fi descriși cu ajutorul mijloacelor oferite de limbajul PASCAL: instrucțiuni, funcții, proceduri și programe ce pot fi derulate pe calculator.

Complexitatea algoritmului se caracterizează prin *necesarul de memorie* și *durata de execuție*. Metodele de estimare a acestor indicatori se studiază într-un compartiment special al informaticii, denumit **analiza algoritmilor**. În cadrul acestui compartiment se utilizează următoarele notații:

n – un număr natural ce caracterizează mărimea datelor de intrare ale unui algoritm. În majoritatea cazurilor n reprezintă numărul de elemente ale unei mulțimi, gradul unei ecuații, numărul de componente ale unui tablou etc.;

$V(n)$ – volumul de memorie internă necesară pentru păstrarea datelor cu care operează algoritmul;

$T(n)$ – timpul necesar executării algoritmului. De obicei, în acest timp nu se include durata operațiilor de introducere a datelor inițiale și de extragere a rezultatelor.

Evident, volumul de memorie $V(n)$ și timpul de execuție $T(n)$ depind, în primul rând, de caracteristica n a datelor de intrare, fapt accentuat și prin folosirea notațiilor ce reprezintă funcții de argumentul n .

Aplicarea practică a unui algoritm este posibilă numai atunci când necesarul de memorie și timpul cerut nu încalcă restricțiile impuse de mediul de programare și capacitatea de prelucrare a calculatorului utilizat.

De exemplu, presupunem că pentru rezolvarea unei probleme există doi algoritmi diferiți, notați prin A_1 și A_2 . Necesarul de memorie și timpul cerut de algoritmul A_1 este:

$$\begin{aligned}V_1(n) &= 100n^2 + 4; \\T_1(n) &= n^3 \cdot 10^{-3},\end{aligned}$$

iar de algoritmul A_2 :

$$\begin{aligned}V_2(n) &= 10n + 12; \\T_2(n) &= 2^n \cdot 10^{-6}.\end{aligned}$$

În aceste formule volumul de memorie se calculează în octeți, iar timpul – în secunde.

Necesarul de memorie și timpul cerut de algoritmi A_1 , A_2 pentru diferite valori ale lui n este prezentat în *tabelul 4.1*.

Tabelul 4.1.

Complexitatea algoritmilor A_1 și A_2

n	10	20	30	40	50
$V_1(n)$	9,77 Kocteți	39,06 Kocteți	87,89 Kocteți	156,25 Kocteți	244,14 Kocteți
$V_2(n)$	112 octeți	212 octeți	312 octeți	412 octeți	512 octeți
$T_1(n)$	1 secundă	8 secunde	9 secunde	16 secunde	25 secunde
$T_2(n)$	0,001 secunde	1,05 secunde	18 secunde	13 zile	36 ani

Din *tabelul 4.1* se observă că algoritmul A_2 devine practic inutilizabil pentru datele de intrare caracteristica cărora $n > 30$. Pentru astfel de date timpul de execuție a algoritmului A_1 este mult mai mic, însă necesarul de memorie $V_1(n)$ poate depăși limita impusă de mediul de programare (64 Kocteți pentru variabilele statice din programele Turbo PASCAL 7.0).

Determinarea necesarului de memorie $V(n)$ și a timpului de execuție $T(n)$ prezintă un interes deosebit la etapa de elaborare a algoritmilor și a programelor respective. Evident, anume pe parcursul acestei etape pot fi eliminați din start acei algoritmi care necesită memorii prea mari sau care au un timp de execuție inacceptabil.

Menționăm că existența unor calculatoare cu memorii din ce în ce mai performante fac ca atenția informaticienilor să fie îndreptată în special asupra necesarului de timp sau, cu alte cuvinte, asupra **complexității temporale** a algoritmilor.

Întrebări și exerciții

- Explicați termenul *complexitatea algoritmului*. Numiți indicatorii ce caracterizează complexitatea algoritmilor.
- Cum credeți, care factori influențează complexitatea unui algoritm?
- De ce depinde necesarul de memorie și timpul cerut de un algoritm? Când este posibilă aplicarea practică a unui algoritm?
- Algoritmii A_1 și A_2 (vezi *tabelul 4.1*) vor derula în mediul de programare Turbo PASCAL 7.0. Cum credeți, care algoritm trebuie utilizat în cazul datelor de intrare cu caracteristica: a) $n = 10$; b) $n = 20$; c) $n = 30$? Pentru care valori ale lui n algoritmul A_1 poate fi utilizat în mediul de programare Turbo PASCAL 7.0?
- Complexitatea unui algoritm, notat prin A_3 , se caracterizează prin

$$V_3(n) = 600n^3 + 18;$$

$$T_3(n) = 3^n \cdot 10^{-2}.$$

Cum credeți, pentru care valori ale lui n algoritmul A_3 poate derula în mediul de programare Turbo PASCAL 7.0?

- ⑥ Determinați mărimea datelor de intrare a celor mai reprezentativi algoritmi elaborați de dvs. în procesul studierii limbajului de programare PASCAL.

4.2. Estimarea necesarului de memorie

Evaluarea necesarului de memorie $V(n)$ poate fi făcută însumând numărul de octeți alocați pentru fiecare variabilă din program. Numărul de octeți alocat unei variabile nestructurate – *integer*, *real*, *boolean*, *char*, *enumerare*, *subdomeniu*, *referință* – depinde de implementarea limbajului. Numărul de octeți alocat unei variabile depinde de implementarea limbajului. În mediul de programare Turbo PASCAL 7.0 memoria se alocă conform tabelului 4.2.

Tabelul 4.2

Alocarea memoriei interne în Turbo PASCAL 7.0

Tipul variabilei	Numărul de octeți
<i>integer</i>	2
<i>real</i>	6
<i>boolean</i>	1
<i>char</i>	1
<i>enumerare</i>	1
<i>subdomeniu</i>	conform tipului de bază
<i>referință</i>	4
<i>pointer</i>	4

În cazul tipurilor structurate de date volumul de memorie necesar unei variabile se calculează însumând numărul de octeți alocați pentru fiecare componentă.

De exemplu, necesarul de memorie pentru variabilele *A*, *B*, *p* și *s* din declarațiile:

```
var A : array[1..n, 1..n] of real;  
    B : array[1..n] of integer;  
    p : boolean;  
    s : string[10];
```

este

$$V(n) = 6n^2 + 2n + 11 \text{ (octeți)}.$$

În general, necesarul de memorie al unui program PASCAL depinde nu numai de tipul variabilelor utilizate, dar și de modul de gestionare a memoriei interne a calcu-

latorului. În procesul derulării unui program PASCAL spațiul de memorie internă este divizat în trei secțiuni (fig. 4.1):

- **segmentul date**, destinat alocării variabilelor globale. Aceste variabile se declară în secțiunea **var** a programului PASCAL;
- **stiva**, destinată alocării parametrilor actuali, variabilelor locale, valorilor returnate de funcții și adreselor de revenire pe durata execuției subprogramelor PASCAL. Apelul unui subprogram implică depunerea datelor respective în stivă, iar ieșirea din subprogram – eliminarea lor. Accentuăm că în cazul parametrului-variabilă în stivă se depune numai adresa variabilei din programul apelant, iar în cazul parametrului-valoare în stivă va fi depusă o copie a datelor din lista parametrilor actuali.
- **heap-ul**, utilizat pentru alocarea variabilelor dinamice. Aceste variabile sînt create și, eventual, distruse cu ajutorul procedurilor `new` și `dispose`.

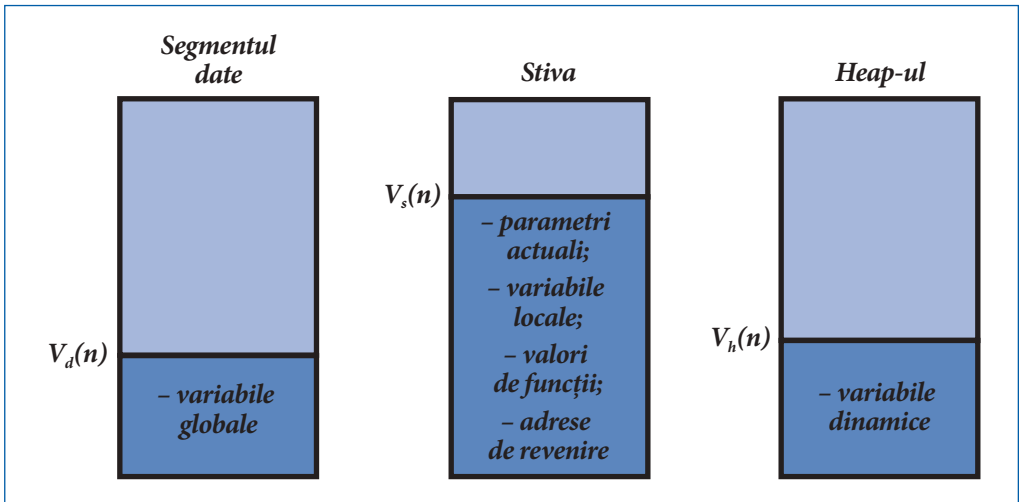


Fig. 4.1. Gestionarea memoriei interne

Prin urmare, estimarea necesarului de memorie presupune evaluarea următoarelor caracteristici ale unui program (fig. 4.1):

$V_d(n)$ – volumul de memorie ocupat de variabilele globale în *segmentul date*;

$V_s(n)$ – volumul de memorie ocupat de parametrii actuali și de variabilele locale în *stivă*;

$V_h(n)$ – volumul de memorie ocupat de variabilele dinamice în *heap*.

De obicei, în mediul de programare Turbo PASCAL 7.0 se cere ca $V_d(n) \leq 64$ Kocteți, $V_s(n) \leq 16$ Kocteți și $V_h(n) \leq 256$ Kocteți. Dimensiunile *stivei* și ale *heap-ului* pot fi modificate cu ajutorul directivelor de compilare sau a comenzilor mediului de programare.

Exemplu:

```

Program P145;
{ Gestionarea memoriei interne }
const n = 100;
type Matrice = array[1..n, 1..n] of real;
      Vector = array[1..n] of real;
    
```



```

var A : Matrice;
      i : integer;
      p, q : ^Matrice;

procedure Prelucrare(var B:Matrice);
var C : Vector;
begin
  {...prelucrarea elementelor matricei B...}
end; { Prelucrare }

begin
  {...introducerea matricei A...}
  Prelucrare(A);
  new(p);
  new(q);
  {...prelucrarea variabilelor dinamice p^ si q^...}
  dispose(p);
  dispose(q);
  {...afișarea rezultatelor...}
  writeln('Sfîrșit');
  readln;
end.

```

Variabilele globale A , i , p și q din programul P145 vor fi depuse în *segmentul date* (fig. 4.2). Necessarul de memorie pentru aceste variabile:

$$V_d(n) = 6n^2 + 2 + 2 \cdot 4 = 6n^2 + 10.$$

Execuția instrucțiunii apel de procedură $Pr(A)$ implică depunerea în stivă a adresei matricei A , a adresei de revenire în programul principal și a variabilei locale C . Necessarul de memorie pentru aceste date:

$$V_s(n) = 6n + 8.$$

După ieșirea din procedură, datele respective vor fi eliminate din stivă.

Instrucțiunile $new(p)$ și $new(q)$ creează în *heap* variabilele dinamice $p^$ și $q^$ de tipul *Matrice*. Necessarul de memorie pentru aceste variabile:

$$V_h(n) = 6n^2 + 6n^2 = 12n^2.$$

După execuția instrucțiunilor $dispose(p)$ și $dispose(q)$, variabilele dinamice din *heap* sînt distruse, iar spațiul respectiv de memorie devine liber.

Întrebări și exerciții

- 1 Determinați cu ajutorul sistemului de asistență al mediului de programare cu care lucrați dvs. necesarul de memorie pentru variabilele nestructurate.
- 2 Cum se evaluează volumul de memorie internă necesar pentru înmagazinarea datelor unui algoritm?

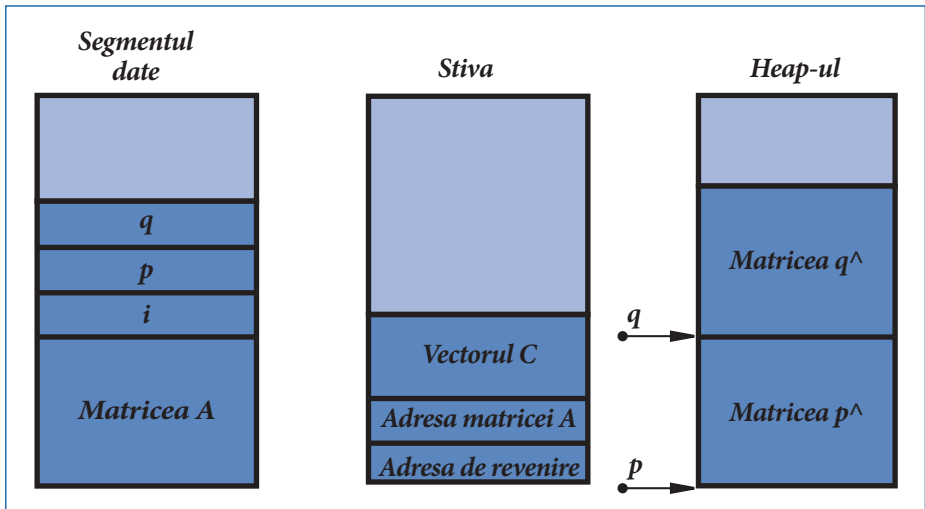


Fig. 4.2. Gestionarea memoriei în programul P145

- ③ Explicați cum se gestionează memoria internă în cazul unui program PASCAL.
- ④ Determinați cu ajutorul sistemului de asistență dimensiunile *segmentului date*, ale *stivei* și ale *heap-ului*. Cum pot fi modificate dimensiunile stivei și ale *heap-ului*?
- ⑤ Calculați necesarul de memorie pentru variabilele din următoarele declarații:

a)

```
var A : array[1..n, 1..n] of integer;
    B : string;
    C : array [1..n, 1..n, 1..n] of boolean;
```

b)

```
type Vector = array[1..n] of real;
    Matrice = array[1..n] of Vector;
var A, B, C : Matrice;
    D : Vector;
```

c)

```
type Elev = record
    Nume : string;
    Prenume : string;
    NotaMedie : real
end;
ListaElevi = array[1..n] of Elev;
var A, B, C : ListaElevi;
```

d)

```
type Angajat = record
    NumePrenume : string;
    ZileLucrate : 1..31;
    PlataPeZi : real;
    PlataPeLuna : real
end;
```

```

        ListaDePlata = array[1..n] of Angajat;
var L1, L2 : ListaDePlata;

```

e)

```

type Elev = record
        Nume : string;
        Prenume : string;
        NotaMedie : real
    end;
    FisierElevi = file of Elev;
var FE : FisierElevi;
    E : Elev;
    str : string;
    i, n : integer;;

```

- ⑥ Evaluați necesarul de memorie pentru programul ce urmează. Compilați acest program pentru valorile 50, 60, 70, 80 și 100 ale constantei n . Explicați mesajele afișate pe ecran.

```

Program P146;
    { Dimensiunile segmentului date }
    const n = 50;
    type Matrice = array[1..n, 1..n] of real;
    var A, B : Matrice;
    begin
        {...introducerea datelor...}
        {...prelucrarea matricelor A și B...}
        writeln('Sfârșit');
        readln;
    end.

```

- ⑦ Se consideră următorul program:

```

Program P147;
    { Dimensiunile stivei }
    var n : integer;

    function S(n:integer):real;
    begin
        if n=0 then S:=0
            else S:=S(n-1)+n;
    end; { S }

    begin
        write('n='); readln(n);
        writeln('s=', S(n));
        readln;
    end.

```

În acest program suma

$$S(n) = 0 + 1 + 2 + \dots + n$$

este calculată cu ajutorul funcției recursive

$$S(n) = \begin{cases} 0, & \text{dacă } n = 0; \\ S(n-1) + n, & \text{dacă } n > 0. \end{cases}$$

Estimați necesarul de memorie al programului P147. Determinați valoarea maximală a lui n pentru care programul P147 derulează fără erori.

- ③ Determinați necesarul de memorie al programului ce urmează. Pentru care valori ale lui n programul va derula fără erori?

```
Program P148;
{ Dimensiunile heap-ului }
type Vector = array[1..100] of real;
var p : ^Vector;
    i, n : integer;
begin
  write('n='); readln(n);
  for i:=1 to n do new(p);
  writeln('Sfârșit');
  readln;
end.
```

4.3. Măsurarea timpului de execuție

În cazul programelor deja elaborate timpul $T(n)$ cerut de un algoritm poate fi aflat prin măsurări directe. Vom folosi în acest scop unitatea de program U7:

```
Unit U7;
{ Masurarea timpului }
interface
function TimpulCurent : real;
implementation
uses Dos;
var ore : word;
    minute : word;
    secunde : word;
    sutimi : word;
function TimpulCurent;
begin
  GetTime(ore, minute, secunde, sutimi);
  TimpulCurent:=3600.0*ore+60.0*minute+
    1.0*secunde+0.01*sutimi;
end; { TimpulCurent }
end.
```

Unitatea de program U7 oferă programatorului funcția *TimpulCurent*, care returnează o valoare de tip real – timpul în secunde. Indicațiile ceasului de sistem în ore, minute, secunde și sutimi de secundă se citesc cu ajutorul procedurii *GetTime* din unitatea de program DOS a mediului de programare Turbo PASCAL 7.0.

Pentru exemplificare prezentăm programul P149 în care se măsoară timpul de execuție a procedurii *Sortare*:

```
Program P149;
  { Timpul de execuție a procedurii Sortare }
uses U7;
type Vector = array[1..10000] of real;
var   A   : Vector;
       i, n : integer;
       T1, T2 : real; { timpul in secunde }

procedure Sortare(var A:Vector; n:integer);
  { Sortarea elementelor vectorului A }
var i, j : integer;
       r : real;
begin
  for i:=1 to n do
    for j:=1 to n-1 do
      if A[j]>A[j+1] then
        begin
          r:=A[j];
          A[j]:=A[j+1];
          A[j+1]:=r;
        end;
end; { Sortare }

begin
  write('Dați numarul de elemente n=');
  readln(n);

  { atribuim lui A valoarea (n, n-1, ..., 3, 2, 1) }
  for i:=1 to n do A[i]:=n-i+1;

  T1:=TimpulCurent;
  Sortare(A, n);
  T2:=TimpulCurent;

  writeln('Durata de execuție', (T2-T1):7:2, ' secunde');
  readln;
end.
```

Procedura Sortare ordonează elementele vectorului A prin metoda bulelor. În această metodă vectorul A este parcurs de n ori, la fiecare parcurgere efectuându-se $n-1$ comparații ale elementelor vecine $A[j]$ și $A[j+1]$. Dacă $A[j] > A[j+1]$, elementele vecine își schimbă locul.

Pentru a evita introducerea de la tastatură a unui număr mare de date, în programul P149 vectorului A i se atribuie valoarea inițială

$$A = (n, n-1, n-2, \dots, 3, 2, 1).$$

De exemplu, pentru $n=4$, vectorul inițial va fi

$$A=(4, 3, 2, 1).$$

În procesul sortării avem:

$$\begin{aligned} i = 1, & \quad A = (3, 2, 1, 4); \\ i = 2, & \quad A = (2, 1, 3, 4); \\ i = 3, & \quad A = (1, 2, 3, 4); \\ i = 4, & \quad A = (1, 2, 3, 4). \end{aligned}$$

Timpu de execuție a procedurii Sortare în cazul unui calculator *Pentium* cu frecvența ceasului de sistem 500 MHz este prezentat în tabelul 4.3, iar graficul respectiv – în figura 4.3.

Tabelul 4.3

Timpul de execuție a procedurii Sortare

n	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
$T(n), s$	0,27	1,10	2,47	4,50	7,03	10,16	13,84	18,02	22,85	28,18

Întrebări și exerciții

- ❶ Cum credeți, ce legătură există între timpul necesar execuției unui program PASCAL, frecvența ceasului de sistem și capacitatea de prelucrare a calculatorului?
- ❷ Măsurați timpul de execuție a procedurii Sortare (vezi programul P149) în cazul calculatorului cu care lucrați dvs. Construiți un grafic similar celui din figura 4.3.
- ❸ Reprezentați grafic pe un singur desen timpul de execuție a procedurilor ce urmează.

```
a) procedure N2(n : integer);
var i, j, k : integer;
    r : real;
begin
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to 300 do
        r:=1.0;
      end; { N2 }
```

```
b) procedure N3(n : integer);
var i, j, k : integer;
    r : real;
```

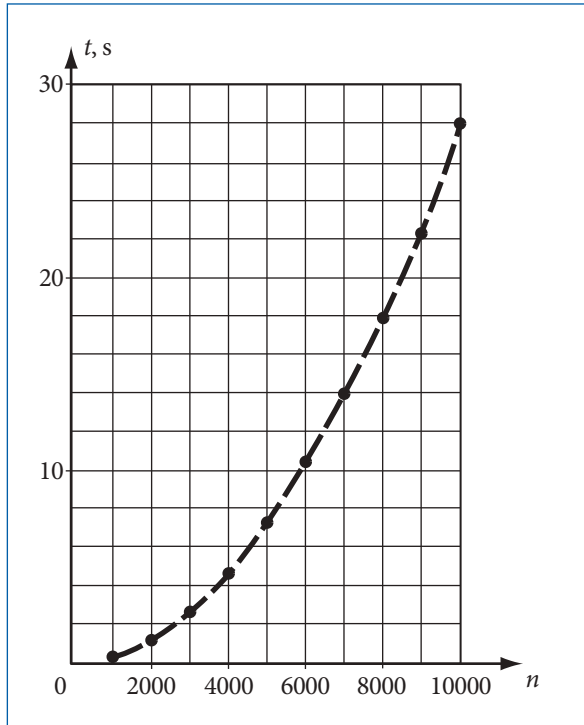


Fig. 4.3. Timpul de execuție a procedurii Sortare

```
begin
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to n do
        r:=1.0;
      end; { N3 }
    end;
  end;
```

c)

```
procedure N4(n : integer);
var i, j, k, m : integer;
    r : real;
begin
  for i:=1 to n do
    for j:=1 to n do
      for k:=1 to n do
        for m:=1 to n do
          r:=1.0;
        end;
      end;
    end;
  end; { N4 }
```

Pentru exemplificare, în figura 4.4 sînt prezentate graficele respective în cazul unui calculator *Pentium*, frecvența ceasului de sistem 500 MHz.

- ④ Care este precizia de măsurare a timpului cu ajutorul funcției `TimpulCurent`? Argumentați răspunsul dvs.

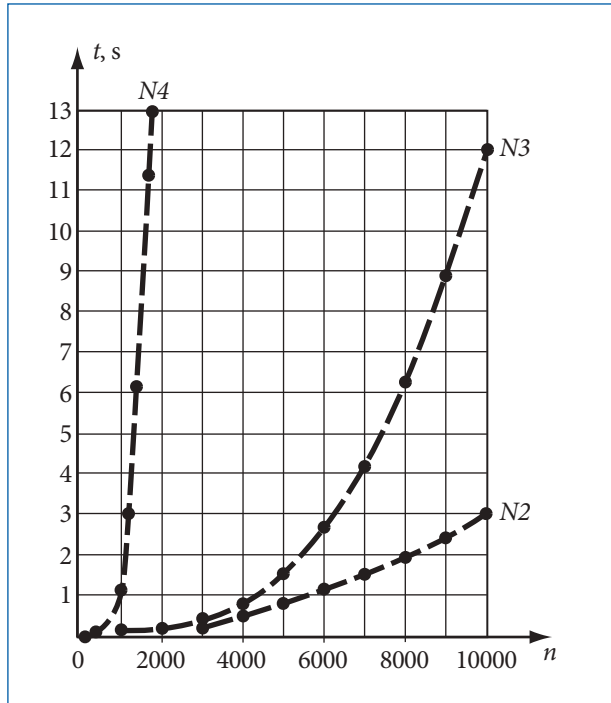


Fig. 4.4. Timpul de execuție a procedurilor N2, N3 și N4

4.4. Estimarea timpului cerut de algoritm

În procesul implementării practice a oricărui algoritm apare necesitatea estimării timpului de execuție $T(n)$ pînă la testarea și depanarea definitivă a programului ce-l realizează. Cu regret, prin metode teoretice este foarte greu de determinat o expresie exactă pentru $T(n)$. Din aceste motive se caută o limită superioară a timpului cerut de algoritm, analizîndu-se doar cazurile cele mai defavorabile.

Presupunem, în scopuri didactice, că execuția oricărui operator PASCAL (+, -, or, *, /, div, and, <, <=, not etc.) necesită cel mult Δ unități de timp. Același timp Δ este necesar pentru indexarea componentelor unui tablou [], pentru atribuirea := și instrucțiunea goto. Valoarea concretă a mărimii Δ depinde de mediul de programare, capacitatea de prelucrare a calculatorului utilizat și este de ordinul $10^{-9} \dots 10^{-7}$ secunde. În continuare vom estima timpul $T(n)$ în forma:

$$T(n) = Q(n) \cdot \Delta,$$

unde $Q(n)$ este numărul de operații elementare – adunarea, scăderea, înmulțirea, compararea etc. – necesare pentru soluționarea unei probleme.

Admitem că într-o expresie E apar m operatori PASCAL și k apeluri ale funcției F . Evident, numărul Q_E de operații elementare necesare pentru calcularea expresiei E se determină ca

$$Q_E = m + k Q_F,$$

unde Q_F este numărul de operații elementare necesare pentru calcularea funcției F .

Exemple:

	<u>Expresia E</u>	<u>Numărul de operații elementare Q_E</u>
a)	$a*b+c$	2
b)	$(a<b) \text{ or } (c>d)$	3
c)	$\sin(x)+\cos(y)$	$1 + Q_{\sin} + Q_{\cos}$
d)	$a+M[i]$	2
e)	$\sin(x+y)+\sin(x-y)$	$3 + 2Q_{\sin}$

Numărul de operații elementare Q_I necesare pentru execuția unei instrucțiuni I a limbajului PASCAL se estimează conform formulelor din tabelul 4.4.

Tabelul 4.4

Numărul de operații elementare necesare pentru execuția unei instrucțiuni PASCAL

Nr. crt.	Instrucțiunea PASCAL	Numărul de operații elementare
1	Atribuirea $v := E$	$Q_E + 1$
2	Apelul procedurii P	$Q_P + 1$
3	Selecție if E then I_1 else I_2	$Q_E + \max\{Q_{I_1}, Q_{I_2}\} + 1$
4	Selecție multiplă case E of $I_{1j}, I_{2j}, \dots, I_{kj}$ end	$Q_E + \max\{Q_{I_{1j}}, Q_{I_{2j}}, \dots, Q_{I_{kj}}\} + k + 1$
5	Ciclu cu contor for $v := E_1$ to/downto E_2 do I	$Q_{E_1} + Q_{E_2} + mQ_I + m + 1$
6	Ciclu cu test inițial while E do I	$(m + 1)Q_E + mQ_I + 1$
7	Ciclu cu test final repeat I until E	$mQ_I + mQ_E + 1$
8	Instrucțiunea compusă begin $I_1; I_2; \dots; I_k$ end	$Q_{I_1} + Q_{I_2} + \dots + Q_{I_k} + 1$
9	Instrucțiunea with v do I	$Q_I + 1$
10	Saltul goto	1

Formulele din tabelul 4.4 pot fi deduse urmînd modul de execuție a fiecărei instrucțiuni PASCAL. În acest tabel v reprezintă o variabilă sau un nume de funcție, E – o expresie, iar I – o instrucțiune. Numărul de execuții ale instrucțiunii I din cadrul unui ciclu **for**, **while** sau **repeat** este notat prin m . Menționăm că ciclurile unui program PASCAL pot fi organizate și cu ajutorul instrucțiunilor **if** și **goto**, însă o astfel de utilizare a acestor instrucțiuni contravine regulilor programării structurate.

Pentru exemplificare vom estima numărul de operații elementare $Q(n)$ necesare ordonării elementelor unui vector prin metoda bulelor:

```

procedure Sortare(var A:Vector; n:integer);
var i, j : integer;
    r : real;
{1} begin
{2}   for i:=1 to n do
{3}     for j:=1 to n-1 do
{4}       if A[j]>A[j+1] then
{5}         begin
{6}           r:=A[j];
{7}           A[j]:=A[j+1];
{8}           A[j+1]:=r;
        end;
end; { Sortare }

```

Instrucțiunile I_1, I_2, \dots, I_8 ale procedurii Sortare vor fi referite cu ajutorul comentariilor {1}, {2}, ..., {8} din partea stângă a liniilor de program. Prin Q_j vom nota numărul de operații elementare necesare pentru executarea instrucțiunii I_j :

$$\begin{aligned}
 Q_6 &= 2; \\
 Q_7 &= 4; \\
 Q_8 &= 3; \\
 Q_5 &= Q_6 + Q_7 + Q_8 + 1 = 10; \\
 Q_4 &= 4 + Q_5 + 1 = 15; \\
 Q_3 &= 0 + 1 + (n-1)Q_4 + (n-1) + 1 = 16n - 14; \\
 Q_2 &= 0 + 0 + nQ_3 + n + 1 = 16n^2 - 13n + 1; \\
 Q_1 &= Q_2 + 1 = 16n^2 - 13n + 2.
 \end{aligned}$$

Prin urmare, numărul de operații elementare

$$Q(n) = 16n^2 - 13n + 2,$$

iar timpul cerut de procedura Sortare

$$T(n) = (16n^2 - 13n + 2)\Delta.$$

Din exemplul studiat mai sus se observă că ordinea parcurgerii instrucțiunilor este impusă de structura programelor PASCAL. Evident, mai întâi se analizează instrucțiunile simple, iar apoi cele structurate. În cazul instrucțiunilor imbricate, mai întâi se analizează instrucțiunile din interior, apoi cele care le cuprind.

Expresiile analitice $T(n)$ obținute în urma analizei programelor PASCAL pot fi folosite pentru determinarea experimentală a timpului Δ necesar efectuării unei operații elementare. De exemplu, pentru procedura Sortare (vezi tabelul 4.3) $n = 10000$ și $T(n) = 28,18$ s. Din ecuația

$$(16n^2 - 13n + 2)\Delta = 28,18$$

obținem $\Delta \approx 1,8 \cdot 10^{-8}$ secunde.

Evident, această valoare este valabilă numai pentru mediul de programare Turbo PASCAL 7.0 și calculatorul *Pentium* cu frecvența ceasului de sistem 500 MHz, utilizate în procesul de măsurare a timpului de execuție a procedurii Sortare. De exemplu, în cazul unui calculator *Pentium* cu frecvența ceasului de sistem 150 MHz se obține valoarea $\Delta \approx 6,0 \cdot 10^{-8}$ secunde.

Întrebări și exerciții

- ❶ Determinați cu ajutorul programului P149 valoarea Δ pentru mediul de programare și calculatorul cu care lucrați dvs.
- ❷ Determinați numărul de operații elementare Q , necesare pentru execuția următoarelor instrucțiuni PASCAL:

a) `x:=2*a-6*(y+z);`

b) `p:=not(a=b)and(c>d);`

c) `p:=(a in R)and(b in P);`

d) `if a>b then x:=0 else x:=a+b;`

e) `case i of
1: x:=0;
2: x:=a+b;
3: x:=a+b+c;
end;`

f) `for i:=1 to n do A[i]:=2*A[i];`

g) `for i:=1 to n do A[i]:=B[i+1]-C[i-2];`

h) `i:=0; while i<n do begin i:=i+1 end;`

i) `i:=n; repeat i:=i-1 until i=0;`

j) `begin i:=0; s:=0; r:=0 end;`

k) `with A do begin x:=0; y:=0 end.`

- ❸ Estimați numărul operațiilor elementare $Q(n)$ din procedurile ce urmează:

a) `procedure N2(n : integer);
var i, j, k : integer;
r : real;
begin
for i:=1 to n do
for j:=1 to n do
for k:=1 to 300 do
r:=1.0;
end; { N2 }`

```

b)  procedure N3(n : integer);
      var i, j, k : integer;
          r : real;
      begin
          for i:=1 to n do
              for j:=1 to n do
                  for k:=1 to n do
                      r:=1.0;
                  end;
              end;
          end; { N3 }
    
```

```

c)  procedure N4(n : integer);
      var i, j, k, m : integer;
          r : real;
      begin
          for i:=1 to n do
              for j:=1 to n do
                  for k:=1 to n do
                      for m:=1 to n do
                          r:=1.0;
                      end;
                  end;
              end;
          end; { N4 }
    
```

- ④ În cazul procedurii *Sortare* pentru un calculator *Pentium* cu frecvența ceasului de sistem $f_1 = 500 \text{ MHz}$ s-a obținut $\Delta_1 \approx 1,8 \cdot 10^{-8} \text{ s}$. Pentru același tip de calculator, însă cu frecvența ceasului de sistem $f_2 = 150 \text{ MHz}$, s-a obținut $\Delta_2 \approx 6,0 \cdot 10^{-8} \text{ s}$. Se observă că

$$\frac{f_1}{f_2} = \frac{500 \cdot 10^6}{150 \cdot 10^6} \approx 3,3 \quad \text{și} \quad \frac{\Delta_2}{\Delta_1} = \frac{6,0 \cdot 10^{-8}}{1,8 \cdot 10^{-8}} \approx 3,3.$$

Cum credeți, prin ce se explică acest fapt?

- ⑤ În procesul compilării, instrucțiunile limbajului PASCAL sînt traduse în una sau mai multe instrucțiuni din limbajul cod-mașină. Numărul concret de instrucțiuni depinde de mediul de programare și tipul calculatorului utilizat. Elaborați planul unui experiment care ne-ar permite să estimăm numărul de instrucțiuni cod-mașină în care este tradusă fiecare instrucțiune PASCAL.
- ⑥ E cunoscut faptul că timpul de execuție a instrucțiunilor din limbajul cod-mașină depinde de tipul lor. De exemplu, o instrucțiune care adună două numere întregi este mai rapidă decît instrucțiunea care adună două numere reale. În consecință, valorile Δ , determinate prin măsurarea timpului de execuție a unui program PASCAL, depind de tipul datelor utilizate. Verificați experimental această afirmație în cazul calculatorului cu care lucrați dvs.
- ⑦ Capacitatea de prelucrare a unui calculator se măsoară în *Mips* – Megainstrucțiuni pe secundă. De exemplu, calculatoarele personale au capacitatea de prelucrare 500–800 *Mips*. Pentru a măsura această caracteristică, producătorii de calculatoare utilizează instrucțiunile limbajului cod-mașină. Evident, pentru un programator PASCAL ar prezenta interes și capacitatea de prelucrare exprimată în instrucțiuni PASCAL pe secundă. Elaborați planul unui experiment care ar permite estimarea acestei caracteristici pentru calculatorul cu care lucrați dvs.

4.5. Complexitatea temporală a algoritmilor

În informatică complexitatea temporală a algoritmilor se caracterizează prin timpul de execuție $T(n)$ sau numărul de operații elementare $Q(n)$. Întrucât calculatoarele moderne au o viteză de calcul foarte mare – $10^8 \dots 10^{10}$ instrucțiuni pe secundă, problema timpului de execuție se pune numai pentru valorile mari ale lui n . În consecință, în formulele ce exprimă numărul de operații elementare $Q(n)$ prezintă interes numai **termenul dominant**, adică acel care tinde cât mai repede la infinit. Importanța termenului dominant față de ceilalți este pusă în evidență în *tabelul 4.5*.

Tabelul 4.5

Valorile termenilor dominanți

n	$\log_2 n$	n^2	n^3	n^4	2^n
2	1	4	8	16	4
4	2	16	64	256	16
8	3	64	512	4096	256
16	4	256	4096	65536	65536
32	5	1024	32768	1048576	4294967296

De exemplu, numărul de operații elementare ale procedurii Sortare se exprimă prin formula:

$$Q(n) = 16n^2 - 13n + 2.$$

Termenul dominant din această formulă este $16n^2$. Evident, pentru valorile mari ale lui n numărul de operații elementare

$$Q(n) \approx 16n^2,$$

iar timpul de execuție

$$T(n) \approx 16n^2 \Delta.$$

În funcție de complexitatea temporală, algoritmi se clasifică în:

- algoritmi polinomiali;
- algoritmi exponențiali;
- algoritmi nederminist polinomiali.

Un algoritm se numește **polinomial** dacă termenul dominant are forma Cn^k , adică

$$Q(n) \approx Cn^k; \quad T(n) \approx Cn^k \Delta,$$

unde:

- n este caracteristica datelor de intrare;
- C – o constantă pozitivă;
- k – un număr natural.

Complexitatea temporală a algoritmilor polinomiali este redată prin notația $O(n^k)$ care se citește „algoritm cu timpul de execuție de ordinul n^k ” sau, mai pe scurt, „**algoritm de ordinul n^k** ”. Evident, există algoritmi polinomiali de ordinul n, n^2, n^3 etc.

De exemplu, algoritmul de sortare a elementelor unui vector prin metoda bulelor este un algoritm polinomial de ordinul n^2 . Acest lucru se observă și din graficul timpului de execuție $T(n)$ a procedurii `Sortare`, grafic prezentat în figura 4.3.

Un algoritm se numește **exponențial** dacă termenul dominant are forma Ck^n , adică

$$Q(n) \approx Ck^n; \quad T(n) \approx Ck^n \Delta,$$

unde $k > 1$. Complexitatea temporală a algoritmilor exponențiali este redată prin notația $O(k^n)$.

Menționăm faptul că tot exponențiali se consideră și algoritmi de complexitate $n^{\log n}$, cu toate că această funcție nu este exponențială în sensul strict matematic al acestui cuvânt.

Din comparația vitezei de creștere a funcțiilor exponențială și polinomială (vezi tabelul 4.5) rezultă că algoritmi exponențiali devin inutilizabili chiar pentru valori nu prea mari ale lui n . Pentru exemplificare amintim tabelul 4.1 în care este prezentat timpul de execuție $T_1(n)$ a unui algoritm polinomial de ordinul $O(n^3)$ și timpul de execuție $T_2(n)$ a unui algoritm exponențial de ordinul $O(2^n)$.

Algoritmi **nederminist polinomiali** se studiază în cursurile avansate de informatică.

În funcție de complexitatea temporală se consideră că o problemă este **ușor rezolvabilă** dacă pentru soluționarea ei există un algoritm polinomial. O problemă pentru care nu există un algoritm polinomial se numește **dificilă**. Accentuăm faptul că această clasificare se referă doar la analiza asimptotică a algoritmilor, adică la comportarea lor pentru valori mari ale lui n . Pentru valorile mici ale lui n situația poate fi diferită.

De exemplu, presupunem că pentru soluționarea unei probleme există doi algoritmi, unul polinomial cu timpul de execuție $T_1(n)$ și altul exponențial cu timpul de execuție $T_2(n)$:

$$T_1(n) = 1000n^2 \Delta;$$

$$T_2(n) = 2^n \Delta.$$

Prin calcule directe se poate verifica că pentru $n = 1, 2, 3, \dots, 18$ timpul $T_2(n) < T_1(n)$. Prin urmare, în situația $n \leq 18$ vom prefera algoritmul exponențial.

În practică, elaborarea programelor de calculator presupune parcurgerea următoarelor etape:

- formularea exactă a problemei;
- determinarea complexității temporale a problemei propuse – problemă ușor rezolvabilă sau problemă dificilă;
- elaborarea algoritmului respectiv și implementarea lui pe un sistem de calcul.

Evident, în cazul unor probleme ușor rezolvabile, programatorul va depune toate eforturile pentru a inventa algoritmi polinomiali, adică algoritmi de ordinul n^k , astfel încât parametrul k să ia valori cât mai mici. În cazul problemelor dificile se va da prioritate algoritmilor care minimizează timpul de execuție cel puțin pentru datele de intrare frecvent utilizate în aplicațiile practice. Metodele de clasificare a problemelor în cele ușor și cele dificil rezolvabile se studiază în cursurile avansate de informatică.

Întrebări și exerciții

❶ Indicați termenii dominanți:

a) $12n + 5$;

b) $6n^2 + 100n + 18$;

c) $15n^3 + 1000n^2 - 25n + 6000$;

d) $2000n^3 + 2^n + 13$;

e) $n^{\log_2 n} + n^5 + 300n^2 + 6$;

f) $3^n + 2^n + 14n^3 + 21$;

g) $n^5 + 10n^4 + 200n^3 + 300n^2 + 1000n$.

❷ Cum se clasifică algoritmii în funcție de complexitatea temporală?

❸ Determinați tipul algoritmilor, cunoscând complexitatea temporală:

a) $Q(n) = 200n + 15$;

b) $Q(n) = 2^n + 25n^2 + 1000$;

c) $Q(n) = n^3 + 3^n + 6000n^2 + 10^6$;

d) $Q(n) = 3^n + 2^n + n^{10} + 4000$

e) $Q(n) = n^{\log_2 n} + n^3 + n^2 + 1500$;

f) $Q(n) = 100n^2 + 15n^3 + 8^n + 900$.

❹ Cum credeți, prin ce se explică importanța termenului dominant în analiza complexității temporale a algoritmilor?

❺ Se consideră procedurile N2, N3 și N4 din paragraful precedent. În urma estimării numărului de operații elementare s-a obținut:

$$Q_{N2}(n) = 602n^2 + 2n + 2;$$

$$Q_{N3}(n) = 2n^3 + 2n^2 + 2n + 2;$$

$$Q_{N4}(n) = 2n^4 + 2n^3 + 2n^2 + 2n + 2.$$

Determinați ordinul de complexitate temporală a algoritmilor descriși cu ajutorul acestor proceduri. Comparați viteza de creștere a timpilor de execuție $T_{N2}(n)$, $T_{N3}(n)$ și $T_{N4}(n)$, folosind în acest scop graficele din figura 4.4.

❻ Se consideră un algoritm format din k cicluri imbricate:

```
for  $i_1 := 1$  to  $n$  do
  for  $i_2 := 1$  to  $n$  do
    ...
      for  $i_k := 1$  to  $n$  do P
```

Numărul de operații elementare Q_p efectuate în procedura P este o mărime constantă. Estimați complexitatea temporală a algoritmului.

- 7 Schițați un algoritm pentru rezolvarea următoarei probleme:
Se consideră mulțimea A formată din n numere întregi. Determinați dacă există cel puțin o submulțime B , $B \subseteq A$, suma elementelor căreia este egală cu m . De exemplu, pentru $A = \{-3, 1, 5, 9\}$ și $m = 7$, o astfel de submulțime există, și anume, $B = \{-3, 1, 9\}$.
Estimați complexitatea temporală a algoritmului elaborat.

TEHNICI DE ELABORARE A ALGORITMILOR

5.1. Iterativitate sau recursivitate

Pe parcursul dezvoltării informaticii s-a stabilit că multe probleme de o reală importanță practică pot fi rezolvate cu ajutorul unor metode standard, denumite **tehnici de programare**: recursia, trierea, metoda reluării, metodele euristice ș.a.

Una din cele mai răspândite tehnici de programare este **recursia**. Amintim că recursia se definește ca o situație în care un subprogram se autoapelează fie direct, fie prin intermediul altui subprogram. Tehnicile în studiu se numesc respectiv **recursia directă** și **recursia indirectă** și au fost studiate în cadrul temei „Funcții și proceduri”.

În general, elaborarea unui program recursiv este posibilă numai atunci când se respectă următoarea **regulă de consistență**: soluția problemei trebuie să fie direct calculabilă ori calculabilă cu ajutorul unor valori direct calculabile. Cu alte cuvinte, definirea corectă a unui algoritm recursiv presupune că în procesul derulării calculului trebuie să existe:

- cazuri elementare, care se rezolvă direct;
- cazuri care nu se rezolvă direct, însă procesul de calcul în mod obligatoriu progresaază spre un caz elementar.

De exemplu, în definiția recursivă a funcției factorial $fact: N \rightarrow N$,

$$fact(n) = \begin{cases} 1, & \text{dacă } n = 0; \\ n \cdot fact(n-1), & \text{dacă } n > 0, \end{cases}$$

deosebim:

- Cazul elementar $n = 0$. În acest caz valoarea $fact(0)$ este direct calculabilă și anume $fact(0) = 1$.

- Cazurile neelementare $n > 0$. În astfel de cazuri valorile $fact(n)$ nu sînt direct calculabile, însă procesul de calcul progresaază către cazul elementar $fact(0)$.

De exemplu, pentru $n = 3$ obținem:

$$fact(3) = 3 \cdot fact(2) = 3 \cdot 2 \cdot fact(1) = 3 \cdot 2 \cdot 1 \cdot fact(0) = 3 \cdot 2 \cdot 1 \cdot 1 = 6.$$

Prin urmare, definiția recursivă a funcției $fact(n)$ este o **definiție consistentă**. Amintim că funcția $fact(n)$ poate fi exprimată în PASCAL, urmînd direct definiția, în forma:

```

function Fact(n:Natural):Natural;
begin
  if n=0 then Fact:=1
  else Fact:=n*Fact(n-1)
end;

```

Modul de gestionare a stivei în cazul apelului Fact (3) este prezentat în figura 5.1.

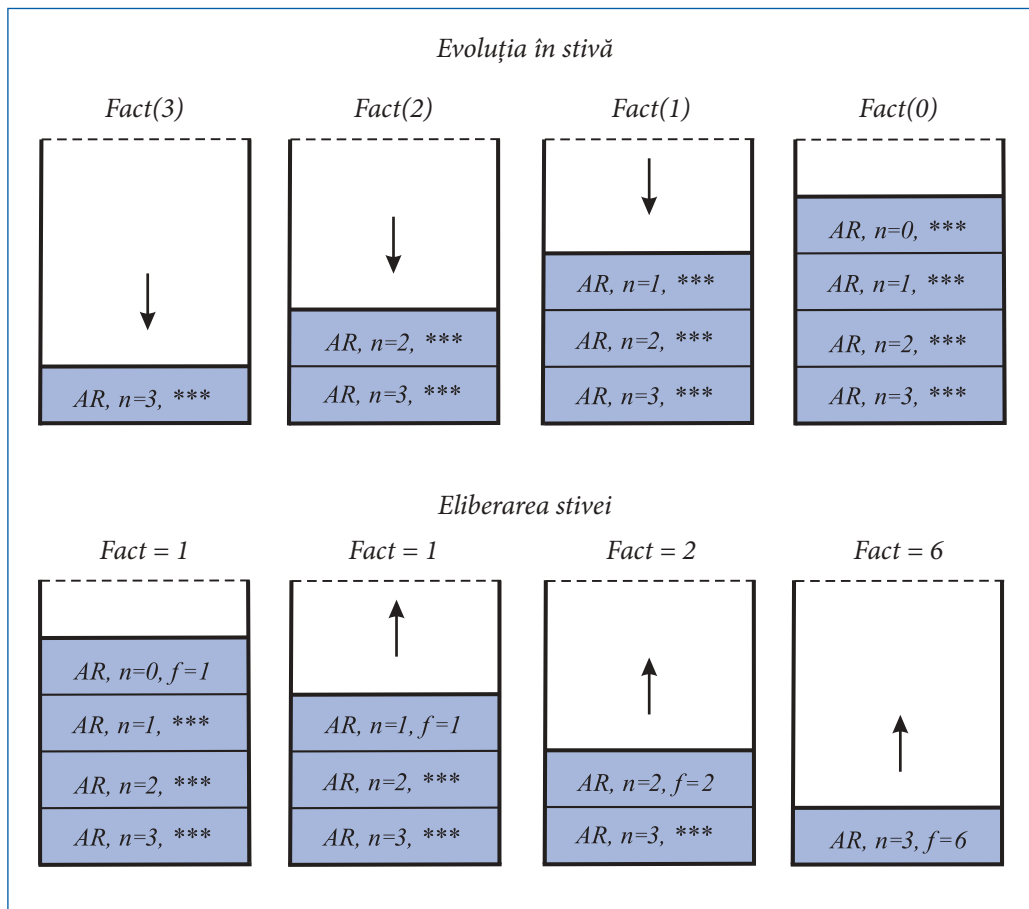


Fig. 5.1. Gestionarea stivei în cazul apelului Fact (3) :
 AR – adresa de revenire; n – valoarea curentă a parametrului actual;
 *** – spațiu pentru memorarea valorii f returnate de funcția Fact

Într-un mod similar, în definiția recursivă a funcției $incons: \mathbb{N} \rightarrow \mathbb{N}$,

$$incons(n) = \begin{cases} 1, & \text{dacă } n = 0; \\ n \cdot incons(n+1), & \text{dacă } n > 0, \end{cases}$$

deosebim cazul elementar $n=0$ și cazurile neelementare $n>0$. Însă, spre deosebire de funcția $fact(n)$, pentru $n > 0$ valorile $incons(n)$ nu pot fi calculate, întrucât procesul de calcul progresează către $incons(\infty)$.

De exemplu, pentru $n=3$ obținem:

$$\text{incons}(3) = 3 \cdot \text{incons}(4) = 3 \cdot 4 \cdot \text{incons}(5) = 3 \cdot 4 \cdot 5 \cdot \text{incons}(6) = \dots$$

Prin urmare, definiția recursivă a funcției $\text{incons}(n)$ este o **definiție inconsistentă** și teoretic procesul de calcul va dura la nesfârșit. În practică calculele vor fi întrerupte de sistemul de operare în momentul depășirii capacității de memorare a stivei sau în cazul depășirii capacității dispozitivului aritmetic.

Accentuăm faptul că mediile actuale de programare nu asigură verificarea consistenței algoritmilor recursivi, acest lucru revenindu-i programatorului.

După cum s-a văzut în capitolele precedente, orice algoritm recursiv poate fi transcris într-un algoritm iterativ și invers. Alegerea tehnicii de programare – **iterativitate** sau **recursivitate** – ține, de asemenea, de competența programatorului. Evident, această alegere trebuie făcută luând în considerare avantajele și neajunsurile fiecărei metode, care variază de la caz la caz. Pentru exemplificare, în *tabelul 5.1* sînt prezentate rezultatele unui studiu comparativ al ambelor tehnici de programare în cazul prelucrării automate a textelor.

Tabelul 5.1

Studiul comparativ al iterativității și recursivității (prelucrarea automată a textelor)

<i>Nr. crt.</i>	<i>Caracteristici</i>	<i>Iterativitate</i>	<i>Recursivitate</i>
1.	Necesarul de memorie	mic	mare
2.	Timpul de execuție	aceiași	
3.	Structura programului	complicată	simplă
4.	Volumul de muncă necesar pentru scrierea programului	mare	mic
5.	Testarea și depanarea programelor	simplă	complicată

Propunem cititorului în calitate de exercițiu efectuarea unui studiu comparativ al iterativității și recursivității în cazul algoritmilor destinați creării și preluării structurilor dinamice de date din *Capitolul 2*.

În general, algoritmii recursivi sînt recomandați în special pentru problemele ale căror rezultate sînt definite prin relații de recurență: analiza sintactică a textelor, prelucrarea structurilor dinamice de date, procesarea imaginilor ș.a. Un exemplu tipic de astfel de probleme este analiza gramaticală a programelor PASCAL, sintaxa căroa, după cum se știe, este definită prin relații de recurență.

Întrebări și exerciții

- 1 Explicați termenul *tehnici de programare*.
- 2 Care este diferența dintre *recursia directă* și *recursia indirectă*? Dați exemple.
- 3 Ce condiții trebuie respectate pentru ca definiția unui algoritm recursiv să fie corectă?
- 4 Care este diferența dintre definițiile recursive consistente și definițiile recursive inconsistente?

- 5 Se consideră următoarele definiții recursive de funcții. Care din aceste definiții sînt consistente? Argumentați răspunsul.

$$a) f: \mathbf{N} \rightarrow \mathbf{N}, \quad f(n) = \begin{cases} 1, & \text{dacă } n = 0; \\ n + f(n-1), & \text{dacă } n > 0; \end{cases}$$

$$b) f: \mathbf{N} \rightarrow \mathbf{N}, \quad f(n) = \begin{cases} 1, & \text{dacă } n = 0; \\ n + f(n), & \text{dacă } n > 0; \end{cases}$$

$$c) f: \mathbf{Z} \rightarrow \mathbf{Z}, \quad f(i) = \begin{cases} 1, & \text{dacă } i = 0; \\ i + f(i-1), & \text{dacă } i \neq 0; \end{cases}$$

$$d) f: \mathbf{N} \rightarrow \mathbf{N}, \quad f(n) = \begin{cases} 1, & \text{dacă } n = 0; \\ n + g(n-1), & \text{dacă } n > 0; \end{cases}$$

$$g: \mathbf{N} \rightarrow \mathbf{N}, \quad g(n) = \begin{cases} 2, & \text{dacă } n = 0; \\ n + f(n-1), & \text{dacă } n > 0; \end{cases}$$

$$e) f: \mathbf{N} \rightarrow \mathbf{N}, \quad f(n) = \begin{cases} n, & \text{dacă } n < 10; \\ (n \bmod 10) + f(n \operatorname{div} 10), & \text{dacă } n \geq 10. \end{cases}$$

- 6 Lansați în execuție programul ce urmează. Explicați mesajele afișate la ecran.

```

Program P150;
  { Depășirea capacității de memorare a stivei }
  type Natural = 0..Maxint;

  function Incons(n:Natural):Natural;
  { Definiție recursivă inconsistentă }
  begin
    writeln('Apel recursiv n=', n);
    if n=0 then Incons:=1
      else Incons:=n*Incons(n+1);
  end; { Incons }

  begin
    writeln(Incons(3));
    readln;
  end.

```

Reprezentați pe un desen similar celui din figura 5.1 modul de gestionare a stivei în cazul apelului *Incons(3)*.

- 7 Indicați cazurile elementare și cele neelementare pentru următorii algoritmi recursivi:
- funcția *Arb* și procedura *AfisArb* din programul P130;
 - procedurile *Preordine*, *Inordine* și *Postordine* din programul P131;
 - procedura *InAdincime* din programul P133.
- 8 Suma primelor n numere naturale $S(n)$ poate fi calculată cu ajutorul funcției iterative

$$S(n) = 0 + 1 + 2 + \dots + n = \sum_{i=0}^n i$$

sau al funcției recursive

$$S(n) = \begin{cases} 0, & \text{dacă } n = 0; \\ n + S(n-1), & \text{dacă } n > 0. \end{cases}$$

Utilizând ca model *tabelul 5.1*, efectuați un studiu comparativ al algoritmilor destinați calculării sumei $S(n)$.

- 9 Se consideră următoarele formule metalingvistice:

$\langle \text{Cifră} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

$\langle \text{Număr} \rangle ::= \langle \text{Cifră} \rangle \{ \langle \text{Cifră} \rangle \}$

$\langle \text{Semn} \rangle ::= + | - | * | /$

$\langle \text{Expresie} \rangle ::= \langle \text{Număr} \rangle | (\langle \text{Expresie} \rangle) | \langle \text{Expresie} \rangle \langle \text{Semn} \rangle \langle \text{Expresie} \rangle$

Scrieți o funcție recursivă care returnează valoarea `true` dacă șirul de caractere S este conform definiției unității lexicale $\langle \text{Expresie} \rangle$ și `false` în caz contrar.

- 10 Efectuați un studiu comparativ al algoritmilor iterativi și algoritmilor recursivi destinați creării și prelucrării următoarelor structuri dinamice de date:

- a) liste unidirecționale;
- b) stiva;
- c) cozi;
- d) arbori binari;
- e) arbori de ordinul m .

- 11 Scrieți o funcție iterativă care returnează valoarea `true` dacă șirul de caractere S este conform definiției unității lexicale $\langle \text{Expresie} \rangle$ din *exercițiul 9* și `false` în caz contrar. Utilizând ca model *tabelul 5.1*, efectuați un studiu comparativ al algoritmului iterativ și algoritmului recursiv.

- 12 Elaborați un subprogram recursiv care calculează suma cifrelor unui număr natural n . Examinați cazurile $n \leq \text{MaxInt}$ și $n \leq 10^{250}$.

- 13 Imaginile în alb-negru (*fig. 5.2*) pot fi codificate cu ajutorul unei matrice binare $B = \|b_{ij}\|_{n \times m}$, $1 \leq n, m \leq 30$. Elementul b_{ij} indică culoarea microzonei respective: neagră ($b_{ij} = 1$) sau albă ($b_{ij} = 0$).

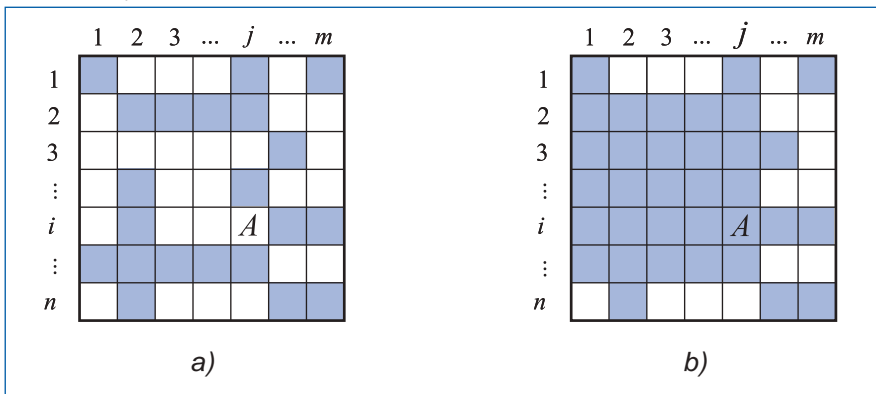


Fig. 5.2. Colorarea unei suprafețe închise:
a – imaginea inițială; b – imaginea finală

Elaborați o procedură recursivă pentru colorarea unei suprafețe închise, specificate prin coordonatele (i, j) ale oricărei microzone A din interiorul ei.

- 14 Elaborați o procedură care determină câte obiecte conține o imagine în alb-negru. Imaginea este împărțită în microzone și codificată cu ajutorul matricei binare $B = \|b_{ij}\|_{n \times m}$. Elementul b_{ij} indică culoarea microzonei cu coordonatele (i, j) .
- 15 Efectuați un studiu comparativ al algoritmilor iterativi și algoritmilor recursivi destinați soluționării problemelor din *exercițiile 13 și 14*.

5.2. Metoda trierii

Metoda trierii presupune că soluția unei probleme poate fi găsită analizând consecutiv elementele s_i ale unei mulțimi finite

$$S = \{s_1, s_2, \dots, s_l, \dots, s_k\},$$

denumită **mulțimea soluțiilor posibile**. În cele mai simple cazuri elementele $s_i, s_i \in S$, pot fi reprezentate prin valori aparținând unor tipuri ordinale de date: *integer*, *boolean*, *char*, *enumerare* sau *subdomeniu*. În problemele mai complicate sîntem nevoiți să reprezentăm aceste elemente prin tablouri, articole sau mulțimi. Menționăm că în majoritatea problemelor soluțiile posibile s_1, s_2, \dots, s_k nu sînt indicate explicit în enunțul problemei și elaborarea algoritmilor pentru calcularea lor cade în sarcina programatorului.

Schema generală a unui algoritm bazat pe metoda trierii poate fi redată cu ajutorul unui ciclu:

```
for i:= 1 to k do
  if SolutiePosibila(si) then PrelucrareaSolutiei(si)
```

unde *SolutiePosibila* este o funcție booleană care returnează valoarea *true* dacă elementul s_i satisface condițiile problemei și *false* în caz contrar, iar *PrelucrareaSolutiei* este o procedură care efectuează prelucrarea elementului selectat. De obicei, în această procedură soluția s_i este afișată la ecran.

În continuare vom analiza două exemple care pun în evidență avantajele și neajunsurile algoritmilor bazați pe metoda trierii.

Exemplul 1. Se consideră numerele naturale din mulțimea $\{0, 1, 2, \dots, n\}$. Elaborați un program care determină pentru câte numere K din această mulțime suma cifrelor fiecărui număr este egală cu m . În particular, pentru $n = 100$ și $m = 2$, în mulțimea $\{0, 1, 2, \dots, 100\}$ există 3 numere care satisfac condițiile problemei: 2, 11 și 20. Prin urmare, $K = 3$.

Rezolvare. Evident, mulțimea soluțiilor posibile $S = \{0, 1, 2, \dots, n\}$. În programul ce urmează suma cifrelor oricărui număr natural $i, i \in S$, se calculează cu ajutorul funcției *SumaCifrelor*. Separarea cifrelor zecimale din scrierea numărului natural i se efectuează de la dreapta la stînga prin împărțirea numărului i și a cîturilor respective la baza 10.

```
Program P151;
{ Suma cifrelor unui număr natural }
type Natural=0..MaxInt;
var i, K, m, n : Natural;
```

```

function SumaCifrelor(i:Natural):Natural;
var suma : Natural;
begin
    suma:=0;
    repeat
        suma:=suma+(i mod 10);
        i:=i div 10;
    until i=0;
    SumaCifrelor:=suma;
end; { SumaCifrelor }

function SolutiePosibila(i:Natural):boolean;
begin
    if SumaCifrelor(i)=m then SolutiePosibila:=true
        else SolutiePosibila:=false;
end; { SumaCifrelor }

procedure PrelucrareaSolutiei(i:Natural);
begin
    writeln('i=', i);
    K:=K+1;
end; { PrelucrareaSolutiei }

begin
    write('Dați n='); readln(n);
    write('Dați m='); readln(m);
    K:=0;
    for i:=0 to n do
        if SolutiePosibila(i) then PrelucrareaSolutiei(i);
    writeln('K=', K);
    readln;
end.

```

Din analiza programului P151 rezultă că complexitatea temporară a algoritmului respectiv este $O(n)$.

Exemplul 2. Se consideră mulțimea $P = \{P_1, P_2, \dots, P_n\}$ formată din n puncte ($2 \leq n \leq 30$) pe un plan euclidian. Fiecare punct P_j este definit prin coordonatele sale x_j, y_j . Elaborați un program care afișează la ecran coordonatele punctelor P_a, P_b , distanța dintre care este maximă.

Rezolvare. Mulțimea soluțiilor posibile $S = P \times P$. Elementele (P_j, P_m) ale produsului cartezian $P \times P$ pot fi generate cu ajutorul a două cicluri imbricate:

```

for j:=1 to n do
    for m:=1 to n do
        if SolutiePosibilă( $P_j, P_m$ ) then PrelucrareaSolutiei( $P_j, P_m$ )

```

Distanța dintre punctele P_j, P_m se calculează cu ajutorul formulei:

$$d_{jm} = \sqrt{(x_j - x_m)^2 + (y_j - y_m)^2}.$$

```
Program P152;
{ Puncte pe un plan euclidian }
const nmax=30;
type Punct = record
    x, y : real;
end;
Indice = 1..nmax;
var P : array[Indice] of Punct;
    j, m, n : Indice;
    dmax : real; { distanța maxima }
    PA, PB : Punct;

function Distanta(A, B : Punct) : real;
begin
    Distanta:=sqrt(sqr(A.x-B.x)+sqr(A.y-B.y));
end; { Distanta }

function SolutiePosibila(j,m:Indice):boolean;
begin
    if j<>m then SolutiePosibila:=true
        else SolutiePosibila:=false;
end; { SolutiePosibila }

procedure PrelucrareaSolutiei(A, B : Punct);
begin
    if Distanta(A, B)>dmax then
        begin
            PA:=A; PB:=B;
            dmax:=Distanta(A, B);
        end;
end; { PrelucrareaSolutiei }

begin
    write('Dati n='); readln(n);
    writeln('Dați coordonatele x, y ale punctelor');
    for j:=1 to n do
        begin
            write('P[', j, ']: '); readln(P[j].x, P[j].y);
        end;
    dmax:=0;

    for j:=1 to n do
        for m:=1 to n do
```



```

if SolutiePosibila(j, m) then
    PrelucrareaSolutiei(P[j], P[m]);

writeln('Soluția: PA=(', PA.x:5:2, ', ', PA.y:5:2, ')');
writeln('          PB=(', PB.x:5:2, ', ', PB.y:5:2, ')');
readln;
end.

```

Complexitatea temporală a algoritmului descris cu ajutorul programului P152 este $O(n^2)$.

Din exemplele prezentate mai sus se observă că în algoritmi bazați pe metoda trierii se calculează, implicit sau explicit, mulțimea soluțiilor posibile S . În problemele relativ simple (*exemplul 1*) elementele din mulțimea soluțiilor posibile pot fi enumerate direct. În problemele mai complicate (*exemplul 2*) generarea soluțiilor posibile necesită elaborarea unor algoritmi speciali. În general, acești algoritmi realizează operațiile legate de prelucrarea unor mulțimi:

- reuniunea;
- intersecția;
- diferența;
- generarea tuturor submulțimilor;
- generarea elementelor unui produs cartezian;
- generarea permutărilor, aranjamentelor sau combinațiilor de obiecte etc.

Avantajul principal al algoritmilor bazați pe metoda trierii constă în faptul că programele respective sînt relativ simple, iar depanarea lor nu necesită teste sofisticate. Complexitatea temporală a acestor algoritmi este determinată de numărul de elemente k din mulțimea soluțiilor posibile S . În majoritatea problemelor de o reală importanță practică metoda trierii conduce la algoritmi exponențiali. Întrucît algoritmi exponențiali sînt inacceptabili în cazul datelor de intrare foarte mari, metoda trierii este aplicată numai în scopuri didactice sau pentru elaborarea unor programe al căror timp de execuție nu este critic.

Întrebări și exerciții

- ❶ Explicați structura generală a algoritmilor bazați pe metoda trierii.
- ❷ Cum poate fi realizată trierea soluțiilor posibile cu ajutorul ciclurilor **while** și **repeat**?
- ❸ Estimați timpul de execuție al programelor P151 și P152. Modificați programul P152 astfel, încît timpul de execuție să se micșoreze aproximativ de două ori.
- ❹ Dați exemple de programe timpul de execuție al cărora nu este critic.
- ❺ Care sînt avantajele și dezavantajele algoritmilor bazați pe metoda trierii?
- ❻ Se consideră mulțimea $P = \{P_1, P_2, \dots, P_n\}$ formată din n puncte ($3 \leq n \leq 30$) pe un plan euclidian. Fiecare punct P_j este definit prin coordonatele sale x_j, y_j . Elaborați un program ce determină trei puncte din mulțimea P pentru care aria triunghiului respectiv este maximă. Estimați timpul de execuție a programului elaborat.
- ❼ Scrieți o funcție PASCAL care, primind ca parametru un număr natural n , returnează valoarea **true** dacă n este prim și **false** în caz contrar. Estimați complexitatea temporală a funcției respective.

- 8 În notația $(a)_x$, litera x reprezintă baza sistemului de numerație, iar litera a – un număr scris în sistemul respectiv. Elaborați un program care calculează, dacă există, cel puțin o rădăcină a ecuației

$$(a)_x = b,$$

unde a și b sînt numere naturale, iar x este necunoscută. Fiecare cifră a numărului natural a aparține mulțimii $\{0, 1, 2, \dots, 9\}$, iar numărul b este scris în sistemul zecimal. De exemplu, rădăcina ecuației

$$(160)_x = 122$$

este $x = 8$, iar ecuația

$$(5)_x = 10$$

nu are soluții. Estimați complexitatea temporală a programului elaborat.

- 9 Într-o pușculiță se află N monede de diferite valori cu greutatea totală G grame. Greutatea fiecărei monede de o anumită valoare este dată în tabelul ce urmează.

Valoarea monedei, lei	Greutatea monedei, grame
1	1
5	2
10	3
25	4
50	5

Elaborați un program care determină suma minimă S care ar putea să fie în pușculiță.

- 10 Elaborați un program care determină cîte puncte cu coordonate întregi se conțin într-o sferă de rază R cu centrul în originea sistemului de coordonate. Se consideră că R este un număr natural, $1 \leq R \leq 30$. Distanța d dintre un punct cu coordonatele (x, y, z) și originea sistemului de coordonate se determină după formula $d = \sqrt{x^2 + y^2 + z^2}$.

5.3. Tehnica Greedy

Această metodă presupune că problemele pe care trebuie să le rezolvăm au următoarea structură:

– se dă o mulțime $A = \{a_1, a_2, \dots, a_n\}$ formată din n elemente;

– se cere să determinăm o submulțime B , $B \subseteq A$, care îndeplinește anumite condiții pentru a fi acceptată ca soluție.

În principiu, problemele de acest tip pot fi rezolvate prin metoda trierii, generînd consecutiv cele 2^n submulțimi A_i ale mulțimii A . Dezavantajul metodei trierii constă în faptul că timpul cerut de algoritmi respectivi este foarte mare.

Pentru a evita trierea tuturor submulțimilor A_i , $A_i \subseteq A$, în metoda *Greedy* se utilizează un **criteriu (o regulă)** care asigură alegerea directă a elementelor necesare din mulțimea A . De obicei, criteriile sau regulile de selecție nu sînt indicate explicit în enunțul problemei și formularea lor cade în sarcina programatorului. Evident, în absența unor astfel de criterii metoda *Greedy* nu poate fi aplicată.

Schema generală a unui algoritm bazat pe metoda *Greedy* poate fi redată cu ajutorul unui ciclu:

```

while ExistaElemente do
begin
  AlegeUnElement(x);
  IncludeElementul(x);
end

```

Funcția `ExistaElemente` returnează valoarea `true` dacă în mulțimea A există elemente care satisfac criteriul (regula) de selecție. Procedura `AlegeUnElement` extrage din mulțimea A un astfel de element x , iar procedura `IncludeElementul` înscrie elementul selectat în submulțimea B . Inițial B este o mulțime vidă.

După cum se vede, în metoda *Greedy* soluția problemei se caută prin testarea consecutivă a elementelor din mulțimea A și prin includerea unora din ele în submulțimea B . Într-un limbaj plastic, submulțimea B încearcă să „înghită” elementele „gustoase” din mulțimea A , de unde provine și denumirea metodei (*greedy* – lacom, hrăpăreț).

Exemplu. Se consideră mulțimea $A = \{a_1, a_2, \dots, a_i, \dots, a_n\}$ elementele căreia sînt numere reale, iar cel puțin unul din ele satisface condiția $a_i > 0$. Elaborați un program care determină o submulțime $B, B \subseteq A$, astfel încît suma elementelor din B să fie maximă. De exemplu, pentru $A = \{21,5; -3,4; 0; -12,3; 83,6\}$ avem $B = \{21,5; 83,6\}$.

Rezolvare. Se observă că dacă o submulțime $B, B \subseteq A$, conține un element $b \leq 0$, atunci suma elementelor submulțimii $B \setminus \{b\}$ este mai mare sau egală cu cea a elementelor din B . Prin urmare, regula de selecție este foarte simplă: la fiecare pas în submulțimea B se include un element pozitiv arbitrar din mulțimea A .

În programul ce urmează mulțimile A și B sînt reprezentate prin vectorii (tablourile unidimensionale) A și B , iar numărul de elemente ale fiecărei mulțimi – prin variabilele întregi, respectiv n și m . Inițial B este o mulțime vidă, respectiv $m=0$.

```

Program P153;
{ Tehnica Greedy }
const nmax=1000;
var A : array [1..nmax] of real;
    n : 1..nmax;
    B : array [1..nmax] of real;
    m : 0..nmax;
    x : real;
    i : 1..nmax;

Function ExistaElemente : boolean;
var i : integer;
begin
  ExistaElemente:=false;
  for i:=1 to n do
    if A[i]>0 then ExistaElemente:=true;
  end; { ExistaElemente }

procedure AlegeUnElement(var x : real);
var i : integer;

```

```

begin
  i:=1;
  while A[i]<=0 do i:=i+1;
  x:=A[i];
  A[i]:=0;
end; { AlegeUnElement }

procedure IncludeElementul(x : real);
begin
  m:=m+1;
  B[m]:=x;
end; { IncludeElementul }

begin
  write('Dați n='); readln(n);
  writeln('Dați elementele mulțimii A:');
  for i:=1 to n do read(A[i]);
  writeln;
  m:=0;
  while ExistaElemente do
    begin
      AlegeUnElement(x);
      IncludeElementul(x);
    end;
  writeln('Elementele mulțimii B:');
  for i:=1 to m do writeln(B[i]);
  readln;
end.

```

Menționăm că procedura `AlegeUnElement` zerografiază componenta vectorului A ce conținea elementul x , extras din mulțimea respectivă.

Complexitatea temporală a algoritmilor bazați pe metoda *Greedy* poate fi evaluată urmînd schema generală de calcul prezentată mai sus. De obicei, timpul cerut de procedurile `ExistaElemente`, `AlegeUnElement` și `IncludeElementul` este de ordinul n . În componența ciclului `while` aceste proceduri se execută cel mult de n ori. Prin urmare, algoritmii bazați pe metoda *Greedy* sînt algoritmi polinomiali. Pentru comparare, amintim că algoritmii bazați pe trierea tuturor submulțimilor A_i , $A_i \subseteq A$, sînt algoritmi de ordinul $O(2^n)$, deci exponențiali. Cu regret, metoda *Greedy* poate fi aplicată numai atunci cînd din enunțul problemei poate fi dedusă regula care asigură selecția directă a elementelor necesare din mulțimea A .

Întrebări și exerciții

- ❶ Explicați structura generală a algoritmilor bazați pe metoda *Greedy*.
- ❷ Care sînt avantajele și neajunsurile algoritmilor bazați pe tehnica *Greedy*?
- ❸ Estimați timpul de execuție al programului P153.

- 4 Schițați un algoritm care determină submulțimea B din exemplul de mai sus prin metoda trierii. Estimați complexitatea temporală a algoritmului elaborat.
- 5 **Memorarea fișierelor pe benzi magnetice.** Se consideră n fișiere f_1, f_2, \dots, f_n care trebuie memorate pe o bandă magnetică. Elaborați un program care determină ordinea de amplasare a fișierelor pe bandă astfel încât timpul mediu de acces să fie minim. Se presupune că frecvența de citire a fișierelor este aceeași, iar pentru citirea fișierului f_i ($i=1, 2, \dots, n$) sînt necesare t_i secunde.
- 6 **Problema continuă a rucsacului.** Se consideră n obiecte. Pentru fiecare obiect i ($i=1, 2, \dots, n$) se cunoaște greutatea g_i și cîștigul c_i care se obține în urma transportului său la destinație. O persoană are un rucsac cu care pot fi transportate unul sau mai multe obiecte greutatea sumară a cărora nu depășește valoarea G_{max} . Elaborați un program care determină ce obiecte trebuie să transporte persoana în așa fel încît cîștigul să fie maxim. În caz de necesitate, unele obiecte pot fi tăiate în fragmente mai mici.
- 7 **Hrubele de la Cricova.** După o vizită la renumitele hrube* de la Cricova un informatician a construit un robot care funcționează într-un cîmp divizat în pătrățele (fig. 5.3). Robotul poate executa doar instrucțiunile SUS, JOS, DREAPTA, STINGA, conform cărora se deplasează în unul din pătrățelele vecine. Dacă în acest pătrat este un obstacol, de exemplu, un perete sau un butoi, are loc un accident și robotul iese din funcțiune.

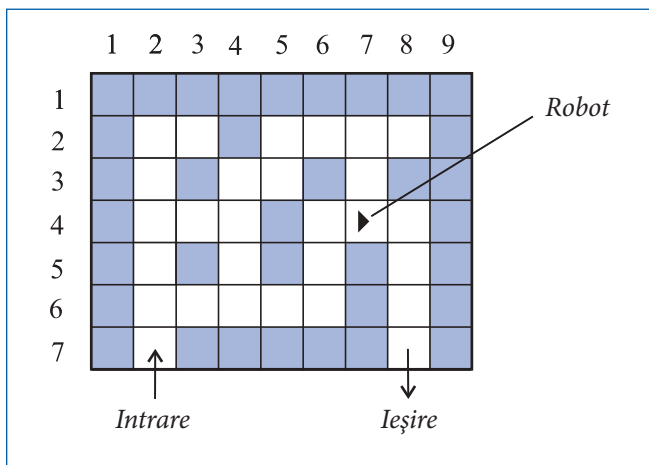


Fig. 5.3. Cîmpul de acțiune al robotului

Elaborați un program care, cunoscînd planul hrubelor, deplasează robotul prin încăperile subterane, de la intrarea în hrube la ieșire. Colecția de vinuri fiind foarte bogată, nu se cere vizitarea obligatorie a tuturor încăperilor subterane.

Datele de intrare. Planul hrubelor este desenat pe o foaie de hîrtie liniată în pătrățele. Pătrățelele hașurate reprezintă obstacolele, iar cele nehașurate – spațiile libere. Pătrățelele de pe perimetrul planului, cu excepția celor de intrare sau ieșire, sînt hașurate prin definiție. În formă numerică planul hrubelor este redat prin tabloul A cu m linii și n coloane. Elementele $A[i, j]$ ale acestui tablou au următoarea semnificație: 0 – spațiu

* *Hrubă* – încăpere sau galerie subterană care servește la depozitarea produselor alimentare. În hrubele de la Cricova pe parcursul mai multor decenii au fost depozitate cele mai bune vinuri din Republica Moldova.

liber; 1 – obstacol; 2 – intrarea în hrube; 3 – ieșirea din hrube. Inițial, robotul se află în pătrățelul pentru care $A[i, j]=2$.

Fișierul HRUBE . IN conține pe prima linie numerele m, n separate prin spațiu. Pe următoarele m linii se conțin câte n numere $A[i, j]$ separate prin spațiu.

Datele de ieșire. Fișierul HRUBE . OUT va conține pe fiecare linie câte una din instrucțiunile SUS, JOS, DREAPTA, STINGA scrise în ordinea executării lor de către robot.

Restricții. $5 \leq m, n \leq 100$. Timpul de execuție nu va depăși 3 secunde.

Exemplu:

HRUBE . IN	HRUBE . OUT
<pre> 7 9 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 1 0 0 0 1 0 0 0 1 1 0 1 0 1 0 1 0 1 1 0 0 0 0 0 1 0 1 1 2 1 1 1 1 1 3 1 </pre>	<pre> SUS DREAPTA DREAPTA DREAPTA DREAPTA SUS SUS DREAPTA DREAPTA JOS JOS JOS </pre>

Indicații. La fiecare pas selectați din mulțimea {SUS, JOS, DREAPTA, STINGA} câte o instrucțiune în așa fel încît robotul să se deplaseze numai de-a lungul unui perete.

5.4. Metoda reluării

În metoda reluării se presupune că soluția problemei pe care trebuie să o rezolvăm poate fi reprezentată printr-un vector:

$$X=(x_1, x_2, \dots, x_k, \dots, x_n).$$

Fiecare componentă x_k a vectorului X poate lua valori dintr-o anumită mulțime $A_k, k = 1, 2, \dots, n$. Se consideră că cele m_k elemente ale fiecărei mulțimi A_k sînt **ordonate** conform unui criteriu bine stabilit, de exemplu, în ordinea apariției lor în memoria calculatorului.

În principiu, astfel de probleme pot fi soluționate prin metoda trierii, tratînd vectorul X ca un element al produsului cartezian $S = A_1 \times A_2 \times \dots \times A_n$. Dezavantajul metodei trierii constă în faptul că timpul cerut de algoritmul respectiv este foarte mare. De exemplu, pentru mulțimile A_1, A_2, \dots, A_n formate numai din câte două elemente, timpul necesar este $O(2^n)$, deci exponențial.

Pentru evitarea trierii tuturor elementelor produsului cartezian $A_1 \times A_2 \times \dots \times A_n$, în metoda reluării componentele vectorului X primesc valori pe rînd, în sensul că lui x_k i se atribuie o valoare numai dacă au fost deja atribuite valori lui x_1, x_2, \dots, x_{k-1} . Mai mult, odată o valoare pentru x_k stabilită, nu se trece direct la atribuirea de valori lui x_{k+1} , ci se verifică anumite **condiții de continuare** referitoare la x_1, x_2, \dots, x_k . Aceste

condiții stabilesc situațiile în care are sens să trecem la calculul lui x_{k+1} . Dacă aceste condiții nu sînt satisfăcute, va trebui să facem o altă alegere pentru x_k sau, dacă elementele din mulțimea A_k s-au epuizat, să micșorăm pe k cu o unitate încercînd să facem o nouă alegere pentru x_{k-1} .

Menționăm faptul că anume micșorarea lui k dă nume metodei studiate, cuvîntul „reluare” semnificînd revenirea la alte variante de alegere a variabilelor x_1, x_2, \dots, x_{k-1} . Evident, aceeași semnificație o are și denumirea engleză a metodei în studiu – *back-tracking* (*back* – înapoi, *track* – urmă).

Pentru exemplificare, în *figura 5.4* este prezentată ordinea în care sînt examinate elementele mulțimilor $A_1 = \{a_{11}, a_{12}\}$, $A_2 = \{a_{21}, a_{22}\}$ și $A_3 = \{a_{31}, a_{32}, a_{33}\}$. În scopuri didactice se consideră că mulțimile A_1 și A_2 au câte două elemente ($m_1 = 2, m_2 = 2$), iar mulțimea A_3 – trei elemente ($m_3 = 3$). Elementele a_{kj} ale mulțimilor respective sînt simbolizate prin ceruțele. Rezultatele verificării condițiilor de continuare sînt reprezentate prin valorile binare 0 (*false*) și 1 (*true*).

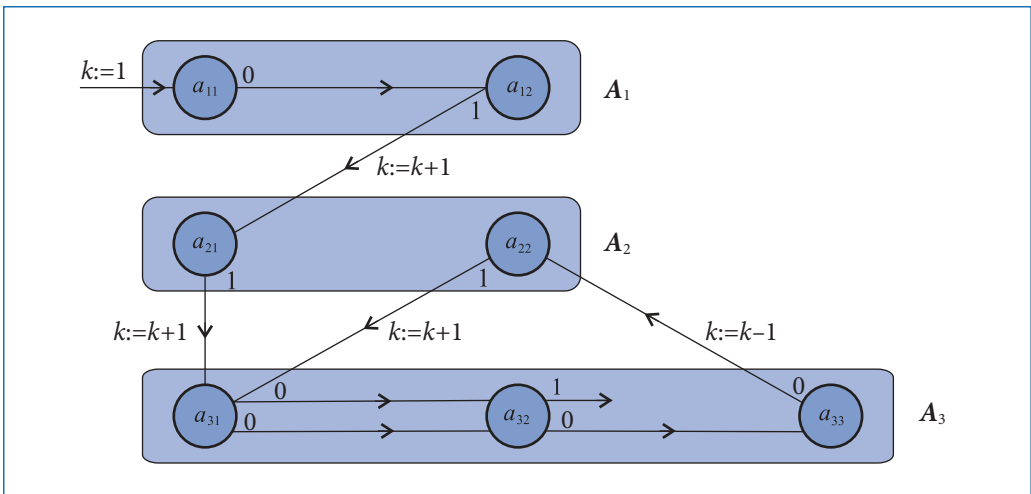


Fig. 5.4. Căutarea soluției prin metoda reluării

Din *figura 5.4* se observă că primul element a_{11} din mulțimea A_1 nu satisface condițiile de continuare și, în consecință, se trece la elementul al doilea a_{12} din aceeași mulțime. Mai departe în vectorul X se include primul element a_{21} din mulțimea A_2 , element care satisface condițiile de continuare, și se trece la examinarea elementelor din mulțimea A_3 .

Întrucît niciunul din elementele a_{31}, a_{32}, a_{33} nu satisface condițiile de continuare, se revine la mulțimea A_2 din care se alege elementul al doilea, și anume, a_{22} . După aceasta se testează din nou elementele mulțimii A_3 , elementul a_{32} satisfăcînd condițiile de continuare.

Schema generală a unui algoritm recursiv bazat pe metoda reluării este redată cu ajutorul procedurii ce urmează:

```

procedure Reluare(k: integer) ;
begin
    if k<=n then

```

```

begin
  X[k]:=PrimulElement(k);
  if Continuare(k) then Reluare(k+1);
  while ExistaSuccesor(k) do
    begin
      X[k]:=Succesor(k);
      if Continuare(k) then Reluare(k+1)
    end; { while }
  end { then }
  else PrelucrareaSolutiei;
end; {Reluare}

```

Procedura Reluare comunică cu programul apelant și subprogramele apelate prin variabilele globale ce reprezintă vectorul X și mulțimile A_1, A_2, \dots, A_n . Subprogramele apelate execută următoarele operații:

PrimulElement(k) – returnează primul element din mulțimea A_k ;

Continuare(k) – returnează valoarea true dacă elementele înscrise în primele k componente ale vectorului X satisfac condițiile de continuare și false în caz contrar;

ExistaSuccesor(k) – returnează valoarea true dacă elementul memorat în componenta x_k are un succesor în mulțimea A_k și false în caz contrar;

Succesor(k) – returnează succesorul elementului memorat în componenta x_k ;

PrelucrareaSolutiei – de obicei, în această procedură soluția reținută în vectorul X este afișată la ecran.

Pentru o înțelegere mai clară a procesului recursiv, vom urmări execuția procedurii Reluare în cazul mulțimilor A_1, A_2 , și A_3 din figura 5.3.

La execuția apelului Reluare(1) în componenta x_1 a vectorului X se înscrie primul element al mulțimii A_1 :

$$X=(a_{11}).$$

Întrucât în cazul acestui vector apelul Continuare(1) returnează valoarea false, se trece la execuția instrucțiunii **while**. Apelul ExistaSuccesor(1) returnează valoarea true și, prin urmare, în componenta x_1 a vectorului X se înscrie succesorul elementului a_{11} :

$$X=(a_{12}).$$

De data aceasta apelul Continuare(1) returnează valoarea true și, prin urmare, se trece la execuția recursivă a apelului Reluare(2). Evident, în componenta x_2 a vectorului X va fi înscris primul element din mulțimea A_2 :

$$X=(a_{12}, a_{21}).$$

Pentru acest vector funcția Continuare returnează valoarea true, fapt ce declanșează execuția recursivă a apelului Reluare(3).

În cazul apelului Reluare(3) componenta x_3 a vectorului X ia recursiv valorile a_{31}, a_{32} și a_{33} :

$$X=(a_{12}, a_{21}, a_{31});$$

$$X=(a_{12}, a_{21}, a_{32});$$

$$X=(a_{12}, a_{21}, a_{33}),$$

însă nici unul din acești vectori nu satisface condițiile de continuare. După examinarea ultimului element din mulțimea A_3 , funcția `ExistăSucesor` returnează valoarea false și, în consecință, se revine în contextul apelului `Reluare (2)`. În acest context în componenta x_2 a vectorului X se înscrie succesorul elementului a_{21} :

$$X=(a_{12}, a_{22}).$$

Întrucât și pentru acest vector funcția `Continuare` returnează valoarea true, din nou se execută apelul recursiv `Reluare (3)`. Însă, spre deosebire de apelul precedent, în acest caz vectorul

$$X=(a_{12}, a_{22}, a_{32})$$

satisface condițiile de continuare, fapt ce declanșează execuția apelului recursiv `Reluare (4)`. În acest apel $k > n$ și, prin urmare, procedura `PrelucrareaSoluției` va afișa coordonatele vectorului X la ecran.

Exemplu. Se consideră mulțimile A_1, A_2, \dots, A_n , fiecare mulțime fiind formată din m_k numere naturale. Selectați din fiecare mulțime câte un număr în așa mod încât suma lor să fie egală cu q .

Rezolvare. Vom reprezenta mulțimile A_1, A_2, \dots, A_n prin liniile tabloului bidimensional (matricei) $A = \|a_{kj}\|$. Numărul de elemente m_k al fiecărei mulțimi A_k va fi reținut în componenta respectivă a tabloului unidimensional (vectorului) $M = \|m_k\|$.

Din enunțul problemei rezultă că toate condițiile de continuare sînt respectate numai atunci cînd suma elementelor referite de primele k componente ale vectorului X nu depășește valoarea q pentru $k < n$ și este egală cu q pentru $k = n$. Pentru a simplifica scrierea programului, vom memora în vectorul X numai indicii j ai elementelor a_{ij} selectate din mulțimile A_1, A_2, \dots, A_n .

```

Program P154;
  { Program bazat pe metoda reluării }
const mmax=50; { numărul maximal de mulțimi }
         nmax=50; { numărul maximal de elemente }
type Natural = 0..MaxInt;
       Multime = array [1..nmax] of Natural;

var   A : array[1..nmax] of Multime;
       n : 1..nmax;           { numărul de mulțimi }
       M : array[1..nmax] of 1..mmax; { cardinalul mulțimii S[k] }
         X : array[1..nmax] of 1..mmax; { indicii elementelor
                                         selectate }

       q : Natural;
       k, j : integer;
       Indicator : boolean;

function PrimulElement(k : integer) : Natural;
begin
  PrimulElement:=1;
end; {PrimulElement }

```

```

function Continuare(k : integer) : boolean;
var j : integer;
    suma : Natural;
begin
    suma:=0;
    for j:=1 to k do suma:=suma+A[j, X[j]];
    if ((k<n) and (suma<q)) or ((k=n) and (suma=q))
        then Continuare:=true
        else Continuare:=false;
end; { Continuare }

function ExistaSuccesor(k : integer) : boolean;
begin
    ExistaSuccesor:=(X[k]<M[k]);
end; { ExistaSuccesor }

function Succesor(k : integer) : integer;
begin
    Succesor:=X[k]+1;
end; { Succesor }

procedure PrelucrareaSolutiei;
var k : integer;
begin
    write('Soluția: ');
    for k:=1 to n do write(A[k, X[k]], ' ');
    writeln;
    Indicator:=true;
end; { PrelucrareaSolutiei }

procedure Reluare(k : integer);
{ Metoda reluării - varianta recursiva }
begin
    if k<=n then
        begin
            X[k]:=PrimulElement(k);
            if Continuare(k) then Reluare(k+1);
            while ExistaSuccesor(k) do
                begin
                    X[k]:=Succesor(k);
                    if Continuare(k) then Reluare(k+1);
                end { while }
            end { then }
            else PrelucrareaSolutiei;
        end; { Reluare }

```

```

begin
    write('Dați cardinalul A[', k, ']='); readln(M[k]);
    write('Dați elementele mulțimii A[', k, ']: ');
    for j:=1 to M[k] do read(A[k, j]);
    writeln;
end;
Write('Dați q='); readln(q);
Indicator:=false;
Reluare(1);
if Indicator=false then writeln('Nu există soluții');
    readln;
end.

```

Din analiza procedurii Reluare și a ordinii de parcurgere a elementelor din mulțimile A_1, A_2, \dots, A_n (fig. 5.4) rezultă că în cazul cel mai nefavorabil vor fi generați toți vectorii X ai produsului cartezian $A_1 \times A_2 \times \dots \times A_n$. Prin urmare, **complexitatea temporală** a algoritmilor bazați pe metoda reluării este, ca și în cazul trierii, $O(m^n)$, unde $m = \max(m_1, m_2, \dots, m_n)$.

Timpul real de execuție, cerut de un algoritm, depinde în mare măsură de natura problemei, mai exact, de esența condițiilor de continuare și ordinea în care apar elementele a_{kj} în mulțimile A_1, A_2, \dots, A_n . O bună alegere a condițiilor de continuare are ca efect o importantă reducere a numărului de calcule. În particular, în cazul cel mai favorabil, din fiecare mulțime A_1, A_2, \dots, A_n va fi testat numai câte un singur element și, evident, timpul de execuție va fi proporțional cu n .

Cu regret, în prezent nu există o regulă universală care ar permite formularea unor condiții „bune” de continuare pentru orice problemă întâlnită în practică. Identificarea acestor condiții și, în consecință, reducerea timpului de calcul depinde de iscusința programatorului. Menționăm că în majoritatea problemelor mulțimile A_1, A_2, \dots, A_n nu sînt cunoscute apriori, iar componentele vectorului X pot aparține și ele unor tipuri structurate de date – tablouri, mulțimi sau articole.

Întrebări și exerciții

- ❶ Explicați structura generală a algoritmilor bazați pe metoda reluării.
- ❷ Indicați cazurile elementare și cele neelementare în procedura recursivă Reluare.
- ❸ Elaborați o variantă iterativă a procedurii Reluare.
- ❹ Elaborați un studiu comparativ al algoritmilor iterativi și algoritmilor recursivi bazați pe metoda reluării.
- ❺ Reprezentați pe un desen similar celui din figura 5.4 ordinea în care sînt examinate elementele mulțimilor A_1, A_2, \dots, A_n din programul P154:
 - a) $A_1=\{1\}, A_2=\{2\}, A_3=\{3\}, q=4$;
 - b) $A_1=\{1\}, A_2=\{2, 3\}, A_3=\{4, 5\}, q=10$;
 - c) $A_1=\{1, 2, 3\}, A_2=\{4, 5\}, A_3=\{6\}, q=14$;
 - d) $A_1=\{1, 2\}, A_2=\{3, 4\}, A_3=\{5, 6\}, A_4=\{7, 8\}, q=20$.

- ⑥ Precizați condițiile de continuare pentru programele în care se dorește generarea tuturor elementelor produsului cartezian $A_1 \times A_2 \times \dots \times A_n$. Pentru a verifica răspunsul scrieți și depanați programul respectiv.
- ⑦ Indicați pe desenul din *figura 5.4* ordinea de parcurgere a elementelor din mulțimile A_1, A_2, A_3 în cazurile cele mai favorabile și cele mai nefavorabile.
- ⑧ Se consideră mulțimea $B = \{b_1, b_2, \dots, b_n\}$ formată în primele n litere ale alfabetului latin. Elaborați un program bazat pe metoda reluării care generează toate submulțimile $B_i, B_i \subseteq B$, formate exact din q litere.

Indicație. Fiecare submulțime B_i poate fi reprezentată prin vectorul caracteristic $X = \|x_k\|_n$, unde

$$x_k = \begin{cases} 1, & \text{dacă } b_k \in B_i; \\ 0, & \text{în caz contrar.} \end{cases}$$

Evident, submulțimea B_i satisface condițiile problemei dacă $x_1 + x_2 + \dots + x_n = q$.

- ⑨ **Colorarea hărților.** Pe o hartă sunt reprezentate n țări, $n \leq 30$. În memoria calculatorului harta este redată prin matricea $A = \|a_{ij}\|_{n \times m}$, unde

$$a_{ij} = \begin{cases} 1, & \text{dacă țările } i, j \text{ sînt vecine;} \\ 0, & \text{în caz contrar.} \end{cases}$$

Determinați numărul minim de culori m necesare pentru a colora harta. Evident, se cere ca oricare două țări vecine să fie colorate diferit.

- ⑩ **Labirintul.** Se consideră planul unui labirint desenat pe o foaie de hîrtie liniată în pătrățele (*fig. 5.5*). Pătrățelele hașurate reprezintă obstacolele, iar cele nehașurate – camerele și coridoarele labirintului.

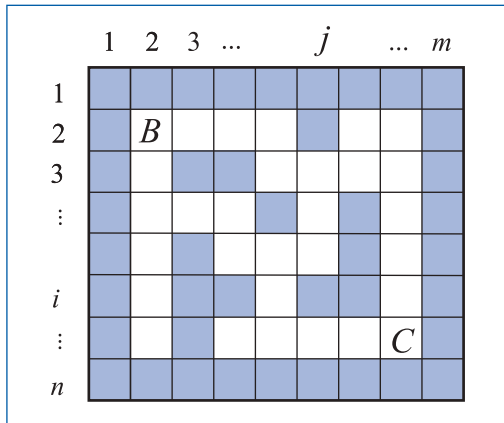


Fig. 5.5. Planul unui labirint

În memoria calculatorului planul labirintului este redat prin matricea $A = \|a_{ij}\|_{n \times m}$, $1 \leq n, m \leq 30$, unde

$$a_{ij} = \begin{cases} 1, & \text{dacă pătrățelele } (i, j) \text{ este hașurat;} \\ 0, & \text{în caz contrar.} \end{cases}$$

Călătorul se poate deplasa dintr-un pătrățel nehașurat în alt pătrățel nehașurat numai atunci cînd ele au o latură comună. Elaborați un program, care găsește, dacă există, un drum din pătrățelul B în pătrățelul C .

- ⑪ Se consideră n ($n \leq 30$) săculețe, numerotate prin $1, 2, 3, \dots, n$. Săculețul i conține m_i monede de aceeași valoare V_i . Elaborați un program care afișează la ecran modul de plată a unei sume S cu exact p monede din săculețe.
- ⑫ Elaborați un program care afișează la ecran toate modurile de a descompune un număr natural în sumă de k numere naturale distincte. De exemplu, pentru $n=9$ și $k=3$ soluțiile sînt: $1+2+6, 2+3+4, 1+3+5$.
- ⑬ Efectuați un studiu comparativ al algoritmilor bazați pe metoda trierii și algoritmilor bazați pe metoda reluării în cazul problemelor din *exercițiile 8 și 12*.

5.5. Metoda *desparte și stăpînește*

Metoda *desparte și stăpînește* (în latină *divide et impera*) este o metodă generală de elaborare a algoritmilor, care presupune:

- 1) împărțirea repetată a unei probleme de dimensiuni mari în două sau mai multe subprobleme de același tip, dar de dimensiuni mai mici;
- 2) rezolvarea subproblemelor în mod direct, dacă dimensiunea lor permite aceasta, sau împărțirea lor în alte subprobleme de dimensiuni și mai mici;
- 3) combinarea soluțiilor subproblemelor rezolvate pentru a obține soluția problemei inițiale.

În limbaj matematic, admitem că la un anumit pas al algoritmului se dă o mulțime ordonată

$$A = (a_i, a_{i+1}, \dots, a_j)$$

și că trebuie efectuată o prelucrare oarecare asupra elementelor sale. Pentru a împărți problema curentă în două subprobleme de aproximativ aceleași dimensiuni, stabilim

$$m = (j - i) \text{ div } 2$$

și împărțim mulțimea A în două submulțimi care vor fi prelucrate separat:

$$A_1 = (a_i, a_{i+1}, \dots, a_{i+m}); \quad A_2 = (a_{i+m+1}, a_{i+m+2}, \dots, a_j).$$

În continuare, mulțimile A_1 și A_2 se împart din nou în câte două submulțimi, respectiv $A_{1,1}, A_{1,2}$ și $A_{2,1}, A_{2,2}$. Acest proces va continua pînă cînd soluțiile ce corespund submulțimilor curente vor putea fi calculate în mod direct.

Pentru exemplificare, în *figura 5.6* este prezentat modul de împărțire a mulțimii $A=(a_1, a_2, \dots, a_7)$ în cazul divizării problemelor curente în câte două subprobleme de același tip.

Schema generală a unui algoritm bazat pe metoda *desparte și stăpînește* poate fi redată cu ajutorul unei proceduri recursive:

```

procedure DesparteSiStapineste(i, j : integer; var x : tip);
var m : integer;
    x1, x2 : tip;
begin
    if SolutieDirecta(i, j) then Prelucrare(i, j, x)

```

```

else
  begin
    m:=(j-i) div 2;
    DesparteSiStapineste(i, i+m, x1);
    DesparteSiStapineste(i+m+1, j, x2);
    Combina(x1, x2, x);
  end;
end;

```

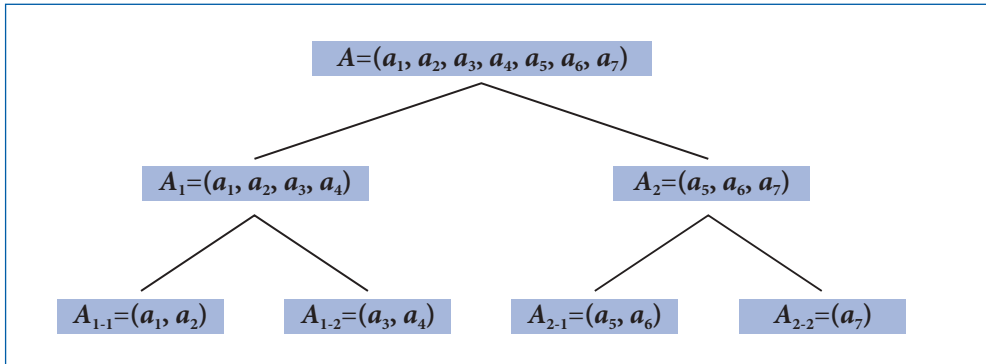


Fig. 5.6. Descompunerea mulțimii A în submulțimi

Parametrii i și j din antetul procedurii definesc submulțimea curentă $(a_i, a_{i+1}, \dots, a_j)$ supusă prelucrării. Funcția *SolutieDirecta* returnează valoarea *true* dacă subproblema curentă admite o rezolvare directă și *false* în caz contrar. Procedura *Prelucrare* returnează prin parametrul x soluția subproblemei curente, calculată în mod direct. Dacă calculul direct este imposibil, se execută două apeluri recursive – unul pentru submulțimea $(a_i, a_{i+1}, \dots, a_m)$ și altul pentru submulțimea $(a_{i+m+1}, a_{i+m+2}, \dots, a_j)$. Procedura *Combina* prelucrează soluțiile x_1 și x_2 ale subproblemelor respective și returnează soluția x a problemei curente.

Exemplul 1. Se consideră mulțimea $A=\{a_1, a_2, \dots, a_n\}$ formată din n numere reale. Elaborați un program care determină numărul maximal din această mulțime.

Rezolvare. În programul ce urmează mulțimea A este reprezentată printr-un tablou unidimensional cu n componente. Se consideră că soluția unei subprobleme poate fi calculată direct numai atunci când mulțimea (a_i, \dots, a_j) este formată din unul sau două numere. Evident, în astfel de cazuri $x = a_i$ sau $x = \max(a_i, a_j)$.

```

Program P155;
{ Gasirea elementului maximal prin metoda desparte și
stăpînește }
const nmax=100;

var A : array[1..nmax] of real;
    i, n : 1..nmax;
    x : real;

```

```

function SolutieDirecta(i, j : integer) : boolean;
begin
    SolutieDirecta:=false;
    if (j-i<2) then SolutieDirecta:=true;
end; { SolutieDirecta }

procedure Prelucrare(i, j : integer; var x : real);
begin
    x:=A[i];
    if A[i]<A[j] then x:=A[j];
end; { Prelucrare }

procedure Combina(x1, x2 : real; var x : real);
begin
    x:=x1;
    if x1<x2 then x:=x2;
end; { Combina }

procedure DesparteSiStapineste(i, j : integer; var x : real);
var m : integer;
    x1, x2 : real;
begin
    if SolutieDirecta(i, j) then Prelucrare(i, j, x)
    else
        begin
            m:=(j-i) div 2;
            DesparteSiStapineste(i, i+m, x1);
            DesparteSiStapineste(i+m+1, j, x2);
            Combina(x1, x2, x);
        end;
end; { DesparteSiStapineste }

begin
    write('Dați n='); readln(n);
    writeln('Dați ', n, ' numere reale');
    for i:=1 to n do read(A[i]);
    writeln;
    DesparteSiStapineste(1, n, x);
    writeln('Numărul maximal x=', x);
    readln;
    readln;
end.

```

În programul P155 la fiecare apel al procedurii DesparteSiStapineste problema curentă este rezolvată direct sau împărțită în două subprobleme de dimensiuni aproximativ egale. Evident, metoda *desparte și stăpânește* admite împărțirea problemelor curente într-un număr arbitrar de subprobleme, nu neapărat în două. În

exemplul ce urmează problema curentă – tăierea unei plăci de arie maximă – este împărțită în patru subprobleme de același tip, dar de dimensiuni mai mici.

Exemplul 2. Se consideră o placă dreptunghiulară de dimensiunile $L \times H$. Placa are n găuri punctiforme, fiecare gaură fiind definită prin coordonatele (x_i, y_i) . Elaborati un program care decupează din placă o bucată de arie maximă, dreptunghiulară și fără găuri. Sînt admise doar tăieturi de la o margine la alta pe direcții paralele cu laturile plăcii – verticale sau orizontale (fig. 5.7).

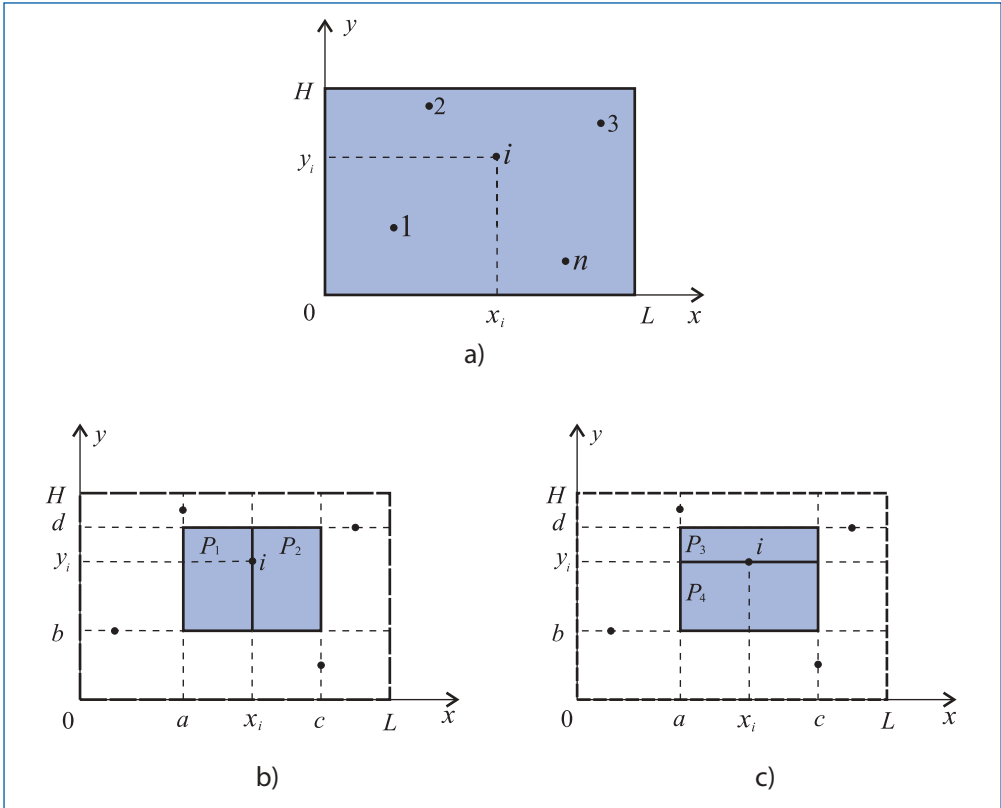


Fig. 5.7. Tăierea unei plăci de arie maximă:
 a – placa inițială; b – tăierea pe verticală; c – tăierea pe orizontală

Rezolvare. Vom defini placa curentă prin vectorul $P=(a, b, c, d)$, unde a și b sînt coordonatele colțului stînga-jos, iar c și d – coordonatele colțului dreapta-sus. Evident, placa inițială se definește prin $(0, 0, L, H)$. Metoda *desparte și stăpînește* poate fi realizată după cum urmează:

- inițial stabilim aria maximă $S_{max}=0$;
- dacă placa curentă nu are găuri, problema poate fi soluționată direct, comparînd aria curentă cu valoarea S_{max} ;
- în caz contrar alegem o gaură arbitrară (x_i, y_i) prin care tăiem placa curentă în plăci mai mici, arătate în figura 5.7:

$$P_1=(a, b, x_i, d), P_2=(x_i, b, c, d) \text{ sau } P_3=(a, y_i, c, d), P_4=(a, b, c, y_i);$$

– în continuare examinăm în același mod fiecare din plăcile obținute în urma tăierii, memorînd consecutiv în variabila S_{max} aria plăcii de suprafață maximă.

```
Program P156;  
{ Tăierea unei placi prin metoda desparte și stăpînește }  
const nmax=100;  
  
var L, H : real;  
    n : 1..nmax;  
    X,Y : array[1..nmax] of real;  
    Smax, amax, bmax, cmax, dmax : real;  
    i : integer;  
  
function SolutieDirecta(a, b, c, d : real;  
                        var i : integer) : boolean;  
  
label l;  
var j : integer;  
begin  
    SolutieDirecta:=true;  
    for j:=1 to n do  
        if (X[j]>a) and (X[j]<c) and (Y[j]>b) and (Y[j]<d) then  
            begin  
                SolutieDirecta:=false;  
                i:=j;  
                goto l;  
            end;  
1:end; { SolutieDirecta }  
  
procedure PrelucrareaSolutiei(a, b, c, d : real);  
var S : real;  
begin  
    S:=(c-a)*(d-b);  
    if S>=Smax then  
        begin  
            Smax:=S;  
            amax:=a; bmax:=b; cmax:=c; dmax:=d;  
        end;  
end; { PrelucrareaSolutiei }  
  
procedure DesparteSiStapineste(a, b, c, d : real);  
var i : integer;  
begin  
    writeln('Examinăm placa (', a:5:1, ' ', b:5:1, ' ',  
            c:5:1, ' ', d:5:1, ')');  
    readln;  
    if SolutieDirecta(a, b, c, d, i)  
        then PrelucrareaSolutiei(a, b, c, d)
```

```

else begin
    DesparteSiStapineste(a, b, X[i], d);
    DesparteSiStapineste(X[i], b, c, d);
    DesparteSiStapineste(a, Y[i], c, d);
    DesparteSiStapineste(a, b, c, Y[i]);
end; { DesparteSiStapineste }

begin
    writeln('Dați dimensiunile L, H'); readln(L, H);
    write('Dați n='); readln(n);
    writeln('Dați coordonatele X[i], Y[i]');
    for i:=1 to n do read(X[i], Y[i]);
    writeln;
    Smax:=0;
    DesparteSiStapineste(0, 0, L, H);
    writeln('Placa de arie maximă (' ,
        amax:5:1, ' ', bmax:5:1, ' ',
        cmax:5:1, ' ', dmax:5:1, ')');
    writeln('Smax=', Smax:5:2);
    readln;
end.

```

Funcția `SolutieDirecta` din programul P156 returnează valoarea `true` dacă placa (a, b, c, d) nu are găuri și `false` în caz contrar. În cazul valorii `false`, funcția returnează suplimentar numărul de ordine i al uneia din găurile plăcii. Această gaură este depistată verificând relațiile $a < x_i < c$ și $b < y_i < d$.

Procedura `PrelucrareaSolutiei` compară aria plăcii curente $S=(c-a)(d-b)$ cu valoarea S_{max} . Dacă $S \geq S_{max}$ procedura memorează placa curentă în vectorul $(a_{max}, b_{max}, c_{max}, d_{max})$.

Complexitatea temporală a algoritmilor bazați pe metoda *desparte* și *stăpînește* depinde de numărul de subprobleme k în care este divizată problema curentă și de complexitatea algoritmilor destinați rezolvării directe a subproblemelor respective. S-a demonstrat că în majoritatea cazurilor timpul cerut de un algoritm bazat pe metoda *desparte* și *stăpînește* este de ordinul $n \log_2 n$ sau $n^2 \log_2 n$, deci polinomial.

Programele elaborate în baza metodei *desparte* și *stăpînește* sînt simple, iar timpul de execuție este relativ mic. Cu regret, această metodă nu este universală, întrucît ea poate fi aplicată numai atunci cînd prelucrarea cerută admite divizarea problemei curente în subprobleme de dimensiuni mai mici. De obicei, această proprietate nu este indicată explicit în enunțul problemei și găsirea ei, dacă există, cade în sarcina programatorului.

Întrebări și exerciții

- ❶ Explicați schema de calcul a algoritmilor bazați pe metoda *desparte* și *stăpînește*.
- ❷ Care sînt avantajele și neajunsurile metodei *desparte* și *stăpînește*?
- ❸ Utilizînd metoda *desparte* și *stăpînește*, elaborați un program care determină suma elementelor mulțimii $A=\{a_1, a_2, \dots, a_n\}$ formată din n numere reale.

- ④ Indicați pe un desen similar celui din *figura 5.7* ordinea examinării plăcilor curente pe parcursul executării programului P156:
- dimensiunea plăcii inițiale: 3×4 ;
 - numărul de găuri ale plăcii inițiale: 3;
 - coordonatele găurilor: $(1, 1)$; $(1, 2)$; $(2, 2)$.
- ⑤ Estimați necesarul de memorie și complexitatea temporală a programelor P155 și P156.
- ⑥ Schițați un algoritm care rezolvă problema tăierii unei plăci de arie maximă prin metoda trierii. Estimați complexitatea temporală și necesarul de memorie al algoritmului elaborat. Efectuați un studiu comparativ al algoritmilor bazați pe metoda trierii și algoritmilor bazați pe metoda *desparte și stăpînește*.
- ⑦ **Căutarea binară.** Se consideră mulțimea $A = \{a_1, a_2, \dots, a_n\}$, elementele căreia sînt numere întregi sortate în ordine crescătoare. Elaborați un program care determină dacă mulțimea A conține numărul dat p . Estimați complexitatea temporală a programului elaborat.
- ⑧ Utilizînd metoda *desparte și stăpînește*, elaborați un program care determină cel mai mare divizor comun al numerelor naturale a_1, a_2, \dots, a_n .
- ⑨ **Sortarea prin interclasare.** Elaborați un program care sortează șirului (a_1, a_2, \dots, a_n) în ordine crescătoare după cum urmează:
- divizăm șirul curent în două subșiruri de aproximativ aceeași lungime;
 - dacă subșirul conține numai două elemente, aranjăm elementele respective în ordine crescătoare;
 - avînd două subșiruri sortate, le interclasăm pentru a obține șirul curent sortat.
- Se consideră că elementele șirului care trebuie sortat sînt numere întregi. De exemplu, în urma interclasării subșirurilor sortate $(-3, 4, 18)$ și $(-2, -1, 15)$ obținem șirul $(-3, -2, -1, 4, 15, 18)$.
- ⑩ **Problema turnurilor din Hanoi.*** Se consideră trei tije notate prin 1, 2 și 3 și n discuri perforate de dimensiuni diferite (*fig. 5.8*). Inițial toate discurile sînt pe tija 1, aranjate în ordinea descrescătoare a diametrelor, considerînd sensul de la bază la vîrf. Elaborați un program care mută toate discurile pe tija 2 folosind tija 3 ca tijă de manevră și respectînd următoarele reguli:
- la fiecare pas se mută un singur disc;
 - orice disc poate fi așezat doar peste un disc cu diametrul mai mare.

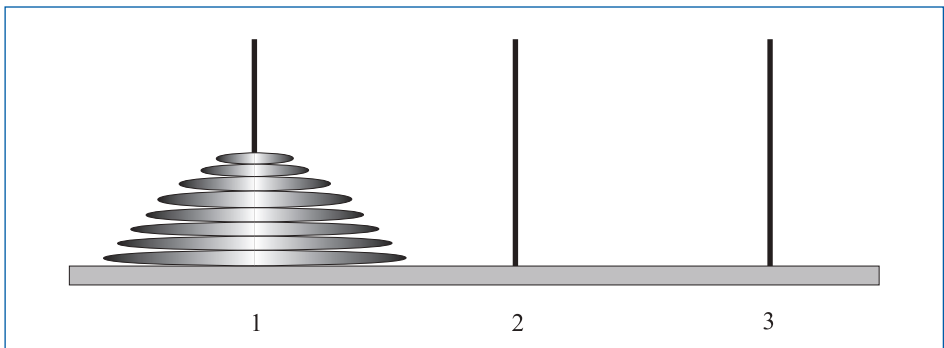


Fig. 5.8. Problema turnurilor din Hanoi

* Denumirea problemei provine de la o veche legendă hindusă conform căreia după mutarea celor 64 de discuri va veni sfîrșitul lumii.

Indicații. Mutarea unui disc de pe tija i pe tija j poate fi reprezentată ca o pereche (i, j) cu $i, j \in \{1, 2, 3\}$, $i \neq j$. Prin $H(m, i, j)$ vom nota șirul mutărilor necesare pentru a muta primele m discuri (evident, cele situate cel mai sus) de pe tija i pe tija j . De exemplu,

$$H(1, 1, 2) = (1, 2);$$

$$H(2, 1, 2) = (1, 3), (1, 2), (3, 2);$$

$$H(3, 1, 2) = (1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2).$$

În general,

$$H(m, i, j) = \begin{cases} (i, j), & \text{pentru } m = 1; \\ H(m-1, i, k), (i, j), H(m-1, k, j), & \text{pentru } m > 1, \end{cases}$$

unde $k = 6 - i - j$. Prin urmare, problema celor n discuri se reduce la rezolvarea a două subprobleme de același tip pentru $(n-1)$ discuri.

5.6. Programarea dinamică

Programarea dinamică reprezintă o metodă de rezolvare a problemelor soluția cărora poate fi privită ca rezultatul unui șir de decizii $(d_1, d_2, \dots, d_p, \dots, d_q)$. Fiecare decizie d_p trebuie să furnizeze o **soluție locală** care optimizează un **criteriu global de calitate**, de exemplu, costul unei călătorii, lungimea drumului parcurs, greutatea transportată, spațiul ocupat de un fișier pe disc etc. Pentru aplicarea acestei metode, este necesar ca problema de rezolvat să satisfacă **principiul optimalității**: dacă $(d_1, d_2, \dots, d_p, d_{p+1}, \dots, d_q)$ este un șir optim de decizii, atunci șirurile (d_1, d_2, \dots, d_p) și (d_{p+1}, \dots, d_q) trebuie să fie optimale.

De exemplu, dacă drumul cel mai scurt dintre *Chișinău* și *Bălți* trece prin *Orhei*, atunci ambele porțiuni din acest drum – *Chișinău-Orhei* și *Orhei-Bălți* – de asemenea vor fi cele mai scurte. Prin urmare, problemele în care se cere determinarea celui mai scurt drum satisfac principiul optimalității.

În PASCAL metoda programării dinamice poate fi realizată utilizând tablouri componentele cărora se calculează cu ajutorul unor **relații de recurență**. În general, relațiile de recurență sînt de următoarele două tipuri:

1) Fiecare decizie d_p depinde de deciziile d_{p+1}, \dots, d_q . Spunem în acest caz că se aplică **metoda înainte**. Deciziile vor fi luate în ordinea d_q, d_{q-1}, \dots, d_1 .

2) Fiecare decizie d_p depinde de deciziile d_1, \dots, d_{p-1} . În acest caz spunem că se aplică **metoda înapoi**. Deciziile vor fi luate în ordinea d_1, d_2, \dots, d_q .

Evident, pentru fiecare problemă propusă, programatorul va verifica mai întîi respectarea principiului optimalității și în cazul unui răspuns pozitiv va deduce relațiile respective de recurență. În caz contrar, problema nu poate fi rezolvată prin metoda programării dinamice.

Exemplu. Planul unui teren aurifer de formă dreptunghiulară cu dimensiunile $n \times m$ este format din zone pătrate cu latura 1 (fig. 5.9). În zona din colțul nord-vest se află un robot. Din zona cu coordonatele (i, j) , $1 \leq i \leq n$, $1 \leq j \leq m$, robotul poate extrage cel mult a_{ij} grame de aur. Din considerente tehnologice pe teren există restricții de deplasare: la fiecare pas robotul se poate mișca din zona curentă numai în una din

zonele vecine – cea din est sau cea din sud. Elaborați un program care determină cantitatea maximă de aur C_{max} care poate fi extrasă de robot.

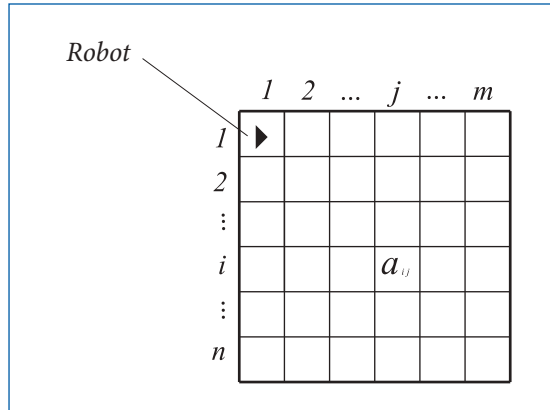


Fig. 5.9. Zonele unui teren aurifer

Rezolvare. În calculator informația despre un teren aurifer poate fi reprezentată cu ajutorul tabloului bidimensional $A = \|a_{ij}\|_{n \times m}$, unde a_{ij} reprezintă cantitatea de aur ce poate fi extrasă din zona cu coordonatele (i, j) . Pentru exemplificare, în continuare vom examina un teren aurifer descris cu ajutorul tabloului ce urmează:

$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 6 & 4 & 6 \\ 3 & 6 & 2 & 7 & 7 & 5 \\ 4 & 7 & 2 & 3 & 4 & 5 \end{array}$$

Pentru a sistematiza calculele, introducem în studiu tabloul bidimensional $C = \|c_{ij}\|_{n \times m}$, unde componenta c_{ij} reprezintă cantitatea maximă de aur pe care o poate extrage robotul deplasându-se din zona inițială $(1,1)$ în zona cu coordonatele (i, j) . Evident, $C_{max} = c_{nm}$.

Conform condițiilor problemei, în oricare zonă (i, j) se poate intra numai prin una din zonele vecine: prin cea din vest cu coordonatele $(i, j-1)$ sau prin cea din nord cu coordonatele $(i-1, j)$. Prin urmare, cantitatea maximă de aur pe care o poate extrage robotul ajungând în zona (i, j) este dată de formula recurentă

$$c_{ij} = a_{ij} + \max(c_{i,j-1}, c_{i-1,j}).$$

Ordinea de calcul a componentelor c_{ij} ale tabloului C este impusă de modul de deplasare a robotului, și anume:

- pasul 1: c_{11} ;
- pasul 2: c_{21}, c_{12} ;
- pasul 3: c_{31}, c_{22}, c_{13} ;
- ...
- pasul $n+m-1$: c_{nm} .

Se observă că la pasul k vor fi calculate numai acele elemente c_{ij} ale tabloului C pentru care se respectă egalitatea $i + j - 1 = k$. Ordinea în care sînt calculate elementele c_{ij} în cazul tabloului A de mai sus este prezentată în figura 5.10.

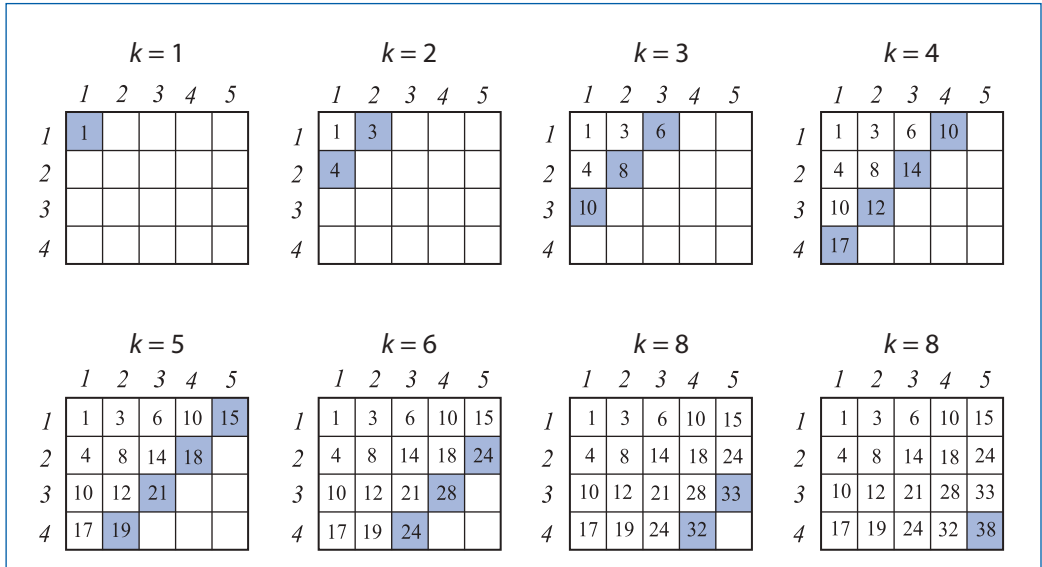


Fig. 5.10. Calcularea elementelor tabloului C

Pentru a evita verificările legate de calcularea indicilor i și j , în programul ce urmează elementele c_{ij} din prima linie și din prima coloană sînt calculate separat de celelalte componente ale tabloului C .

```

Program P157;
{ Deplasarea robotului pe un teren aurifer }
var A , C : array [1..50, 1..50] of real;
    m, n, i, j, k : integer;

function Max(a, b : real) : real;
begin
    if a>b then Max:=a else Max:=b;
end; { Max }

begin
    write('Dați valorile n, m: '); readln(n, m);
    writeln('Dați componentele tabloului A');
    for i:=1 to n do
        for j:=1 to m do read(A[i,j]);
    writeln;

    C[1,1]:=A[1,1];
    for i:=2 to n do C[i,1]:=A[i,1]+C[i-1,1];
    for j:=2 to m do C[1,j]:=A[1,j]+C[1,j-1];
  
```

```

for k:=2 to n+m-1 do
  for i:=2 to n do
    for j:=2 to m do
      if (i+j-1)=k then
        C[i,j]:=A[i,j]+Max(C[i,j-1], C[i-1,j]);
writeln('Cmax=', C[n,m]);
readln;
end.

```

Complexitatea temporală a algoritmilor bazați pe metoda programării dinamice depinde de natura relațiilor de recurență și în majoritatea cazurilor este polinomială. De exemplu, programul P157 are complexitatea $O(n^3)$.

Întrebări și exerciții

- ❶ Explicați esența principiului optimalității.
- ❷ În ce ordine pot fi luate deciziile în procesul soluționării unei probleme prin metoda programării dinamice?
- ❸ Explicați ordinea de luare a deciziilor în problema deplasării robotului pe un teren aurifer.
- ❹ Demonstrați că problema deplasării robotului pe un teren aurifer satisface principiul optimalității.
- ❺ Estimați necesarul de memorie și timpul de execuție al programului P157.
- ❻ Cum credeți, care sînt avantajele și neajunsurile algoritmilor bazați pe metoda programării dinamice?
- ❼ **Problema discretă a rucsacului.** Se consideră n obiecte. Pentru fiecare obiect i ($i=1, 2, \dots, n$) se cunoaște greutatea g_i și câștigul c_i care se obține în urma transportului său la destinație. O persoană are un rucsac cu care pot fi transportate unul sau mai multe obiecte greutatea sumară a cărora nu depășește valoarea G_{max} . Elaborați un program care determină ce obiecte trebuie să transporte persoana în așa fel încît câștigul să fie maxim. Obiectele respective nu pot fi tăiate în fragmente mai mici.
- ❽ **Drumul de cost minim.** Se consideră o rețea formată din n orașe. Între unele orașe există curse directe de autobuz. Costul unui bilet la o cursă directă din orașul i în orașul j este de c_{ij} lei. Evident, costul unei curse directe întotdeauna este mai mic decît costul unei călătorii cu transbordări: $c_{ij} < c_{ik} + c_{kj}$. Elaborați un program care determină costul minim al unei călătorii din orașul i în orașul j .
- ❾ **Arhivarea fișierelor.** E cunoscut faptul că pe suporturile de memorie externă orice fișier este memorat ca o secvență de cifre binare. Pentru a economisi spațiul de memorare, programele de arhivare descompun fișierul inițial într-o succesiune de cuvinte binare ce aparțin unui dicționar și înscriu pe disc numai numerele de ordine ale cuvintelor respective. De exemplu, în cazul unui dicționar format din 5 cuvinte binare:
 - 1) 0;
 - 2) 1;
 - 3) 000;
 - 4) 110;
 - 5) 1101101,

fișierul inițial 10001101101110 va fi memorat pe disc în forma 2354. Elaborați un program care descompune orice secvență binară într-un număr minim de cuvinte aparținând dicționarului propus.

- ⑩ **Triangularea polinoamelor.** Procesarea imaginilor cu ajutorul calculatoarelor presupune descompunerea poligoanelor în figuri geometrice mai simple, și anume, în triunghiuri. Admitem că poligonul convex $P_1P_2\dots P_n$ este definit prin coordonatele (x_i, y_i) ale vîrfurilor $P_i, i=1, 2, \dots, n$. Elaborați un program care descompune poligonul $P_1P_2\dots P_n$ în triunghiuri trasînd în acest scop un set de coarde $P_jP_m, j \neq m, j, m \in \{1, 2, \dots, n\}$, care nu se intersectează. Se cere ca lungimea totală a coardelor respective să fie minimă.

5.7. Metoda ramifică și mărginește

În metoda *ramifică și mărginește* (în engleză *branch and bound*) căutarea soluției unei probleme se efectuează prin parcurgerea unui arbore de ordinul m , denumit **arborele soluțiilor**. Nodurile acestui arbore reprezintă mulțimea stărilor posibile $S = \{s_1, s_2, \dots, s_n\}$ în care se poate afla un sistem, de exemplu, configurațiile pieselor pe o tablă de șah, poziția unui automobil pe harta drumurilor naționale, pozițiile capului de scriere-citire ale unei unități de disc magnetic etc. Legăturile s_i-s_j de tipul tată-fiu indică faptul că din starea s_i sistemul poate trece direct în starea s_j în urma unor transformări sau operații relativ simple (fig. 5.11). Pentru exemplificare amintim deplasarea pieselor făcută alternativ de adversari în cursul unei partide de șah, trecerea automobilului dintr-o localitate în alta, executarea unei comenzi de scriere-citire etc. Evident, în astfel de cazuri soluția problemei poate fi reprezentată ca un drum ce leagă nodul rădăcină cu unul din nodurile terminale și care optimizează un **criteriu de calitate**, de exemplu, numărul de mutări într-o partidă de șah, distanța parcursă de automobil, timpul de scriere-citire a unui fișier etc.

Formal, astfel de probleme pot fi rezolvate prin metoda trierii, efectuînd în acest scop următoarele operații:

- parcurgem arborele soluțiilor în adîncime sau în lățime și găsim stările finale ale sistemului;
- construim mulțimea tuturor drumurilor posibile ce leagă starea inițială cu stările finale;
- din mulțimea drumurilor posibile îl alegem pe cel ce optimizează criteriul de calitate.

Întrucît parcurgerea completă a arborelui necesită foarte mult timp, pentru a micșora numărul de noduri vizitate, în metoda ramifică și mărginește ordinea de parcurgere este impusă de valorile unei funcții $f: S \rightarrow R$, denumită **funcție de cost**. Această funcție este definită pe mulțimea de noduri ale arborelui soluțiilor, valoarea $f(s_i)$ caracterizînd „cheltuielile” necesare pentru a trece din starea curentă s_i în una din stările finale ale subarborelui de rădăcină s_i .

Vizitarea nodurilor unui arbore în ordinea impusă de valorile funcției de cost se numește **parcurgere de cost minim**. Această parcurgere se realizează memorînd nodurile active într-o listă din care la fiecare pas se extrage nodul de cost minim. Pentru comparare amintim că la parcurgerea arborilor în lățime sau în adîncime nodurile active se memorizează într-o coadă sau, respectiv, într-o stivă.

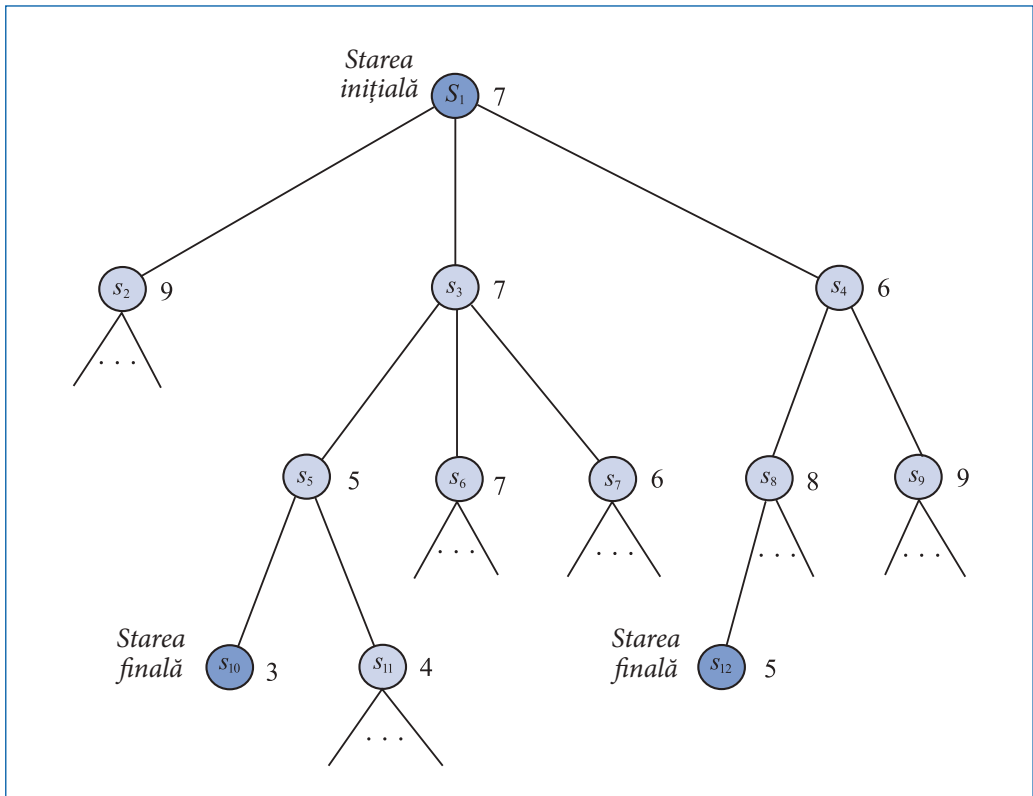


Fig. 5.11. Arborele soluțiilor

Fiind definită o funcție de cost, metoda ramifică și mărginește presupune efectuarea următoarelor operații:

1. *Etapa inițială*: se creează lista nodurilor active. La început această listă conține un singur element – rădăcina arborelui soluțiilor.
2. *Etapa ramifică*: din lista nodurilor active se extrage nodul de cost minim și se generează toți descendenții acestuia.
3. *Etapa mărginește*: pentru fiecare descendent se calculează valoarea funcției de cost. Descendenții costul cărora este inacceptabil (de obicei, subarborii respectivi nu conțin stări finale) sînt excluși din studiu. Descendenții rămași se includ în lista nodurilor active.

Punctele 2 și 3 se repetă pînă cînd se ajunge la una din stările finale sau lista nodurilor active devine vidă. Evident, în ultimul caz soluția problemei nu a fost găsită.

Pentru exemplificare, în tabelul 5.2 este reprezentată ordinea în care sînt vizitate nodurile arborelui din figura 5.11. Valorile funcției de cost sînt indicate direct pe desen lîngă nodurile respective. Se consideră că descendenții costul cărora depășește valoarea 9 sînt inacceptabili.

În cazul arborelui din figura 5.11 parcurgerea de cost minim se termină cînd în lista nodurilor active apare starea finală s_{10} . Întrucît orice drum ce leagă o stare finală de starea inițială poate fi construit urmînd legăturile de tipul fiu-tată, soluția problemei pentru exemplul în studiu este (s_1, s_3, s_5, s_{10}) .

Parcurgerea de cost minim

Nr. crt.	Etapa	Lista nodurilor active	Nodul de cost minim	Descendenții nodului de cost minim
1.	inițială	(s_1)	-	-
2.	ramifică	\emptyset	s_1	s_2, s_3, s_4
3.	mărginește	(s_3, s_4)	-	-
4.	ramifică	(s_3)	s_4	s_8, s_9
5.	mărginește	(s_3, s_8)	-	-
6.	ramifică	(s_8)	s_3	s_5, s_6, s_7
7.	mărginește	(s_8, s_5, s_6, s_7)	-	-
8.	ramifică	(s_8, s_6, s_7)	s_5	s_{10}, s_{11}
9.	mărginește	$(s_8, s_6, s_7, s_{10}, s_{11})$	-	-
10.	stop	(s_8, s_6, s_7, s_{11})	s_{10}	-

Schema de calcul a metodei *ramifică și mărginește* poate fi redată cu ajutorul următoarelor secvențe de instrucțiuni PASCAL:

```

InițializareaListei;
repeat
  Ramifica;
  Margineste;
until Gasit or ListaEsteVida

```

Complexitatea algoritmilor bazați pe metoda *ramifică și mărginește* depinde de faptul cum este definită funcția de cost $f: S \rightarrow R$. În cazul unor definiții nereușite vor fi vizitate toate nodurile arborelui soluțiilor, iar complexitatea temporală a algoritmului respectiv va fi egală cu cea a algoritmilor bazați pe metoda trierii. Din contra, o alegere reușită a funcției de cost permite excluderea din studiu a subarborilor care nu conțin stările finale dorite. În cursurile avansate de informatică se demonstrează că o definiție „bună” a funcției de cost poate fi reprezentată în forma:

$$f(s_i) = \text{niv}(s_i) + h(s_i),$$

unde $\text{niv}(s_i)$ este nivelul nodului s_i , iar funcția $h(s_i)$ estimează complexitatea transformărilor (operațiilor) necesare pentru a ajunge din starea curentă s_i în una din stările finale ale subarborului de rădăcină s_i .

De exemplu, în cazul unei partide de șah funcția $h(s_i)$ poate să exprime numărul de piese grele ale adversarului, în cazul unui automobil – distanța pînă la punctul de destinație, iar în cazul unei unități de disc magnetic – timpul necesar pentru a citi sau a înscrie informația dorită.

Accentuăm faptul că aplicarea practică a metodei *ramifică și mărginește* este posibilă numai atunci cînd valorile $f(s_i)$ ale funcției de cost pot fi calculate fără a construi subarborii de rădăcină s_i . Cu regret, în prezent nu sînt cunoscute reguli universale care ar asigura definirea univocă a funcției de cost pentru orice tip de problemă. Prin

urmare, complexitatea algoritmilor bazați pe metoda *ramifică și mărginește* depinde în mare măsură de experiența și iscusința programatorului.

Întrebări și exerciții

- ❶ Pentru care tip de probleme poate fi aplicată metoda *ramifică și mărginește*?
- ❷ Cum se construiește arborele soluțiilor? Ce informație conține fiecare nod al arborelui?
- ❸ Cum se reprezintă soluția unei probleme în metoda *ramifică și mărginește*?
- ❹ Care este destinația funcției de cost? Cum poate fi definită această funcție?
- ❺ Explicați schema de calcul a metodei *ramifică și mărginește*.
- ❻ Scrieți o procedură PASCAL care realizează parcurgerea de cost minim. Se consideră că fiecare nod al arborelui conține un cîmp în care este indicată valoarea funcției de cost. Estimați necesarul de memorie și complexitatea temporală a procedurii elaborate.
- ❼ Indicați ordinea în care sînt vizitate nodurile arborelui din *figura 5.11* în următoarele cazuri:
 - a) parcurgerea în lățime;
 - b) parcurgerea în adîncime;
 - c) parcurgerea de cost minim.
- ❽ Sînt oare vizitate toate nodurile unui arbore în cazul parcurgerii de cost minim? Argumentați răspunsul dvs.
- ❾ Schițați un algoritm care caută soluția optimă parcurgînd arborele soluțiilor în lățime (în adîncime). Estimați necesarul de memorie și complexitatea temporală a algoritmului elaborat.
- ❿ Dați exemple de probleme ce pot fi soluționate prin metoda *ramifică și mărginește*.
- ⓫ Încercați să desenați nodurile de pe nivelele 0, 1 și 2 ale unui arbore ce reprezintă jocul dvs. preferat, de exemplu, lupta maritimă, șah, dame etc.
- ⓬ Cum credeți, care sînt avantajele și neajunsurile algoritmilor bazați pe metoda *ramifică și mărginește*?

5.8. Aplicațiile metodei *ramifică și mărginește*

Metoda *ramifică și mărginește* poate fi aplicată numai problemelor din enunțul cărora rezultă regulile sau operațiile necesare pentru generarea directă a arborelui soluțiilor. De obicei, în astfel de probleme se simulează comportamentul unor parteneri ce au scopuri sau tendințe contrare: aplicații militare, scenarii de dezvoltare economică în condiții de concurență, jocuri de șah sau de dame etc. Menționăm că existența unor reguli de generare directă a descendenților face inutilă construirea completă a arborelui soluțiilor, micșorîndu-se astfel volumul de memorie internă cerut de algoritm.

Implementarea practică a metodei *ramifică și mărginește* presupune rezolvarea următoarelor probleme:

– determinarea mulțimii de stări care, în funcție de natura problemei de rezolvat, poate fi finită sau infinită;

- stabilirea transformărilor (operațiilor) elementare în urma cărora sistemul trece dintr-o stare în alta;
- definirea funcției de cost;
- definirea structurilor de date necesare pentru a reprezenta stările sistemului, arborele soluțiilor și lista nodurilor active;
- elaborarea subprogramelor necesare pentru generarea descendenților, calcularea funcției de cost, selectarea nodurilor de cost minim etc.

Pentru exemplificare vom analiza modul de implementare a metodei ramifică și mărginește în cazul **jocului Perspico**, cunoscut și sub denumirea **jocul 15**. În acest joc sînt 15 plăcuțe pătrate numerotate de la 1 la 15 (fig. 5.12).

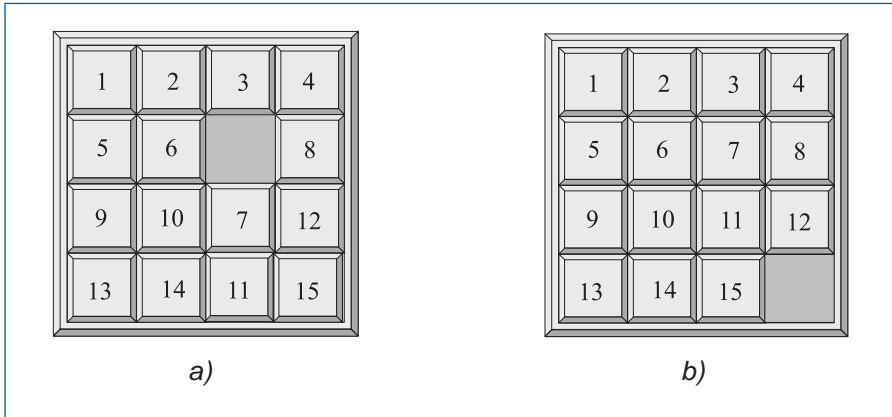


Fig. 5.12. Jocul Perspico:
a – starea inițială; b – starea finală

Plăcuțele sînt încadrate într-o ramă pătrată de dimensiunile 4x4, o poziție din interiorul ramei fiind liberă. Orice plăcuță vecină cu poziția liberă poate fi mutată pe locul respectiv. Se cere a trece, folosind mutările permise, din starea inițială în starea finală.

Starea curentă a jocului Perspico poate fi exprimată printr-o distribuție a celor 15 plăcuțe în cele 16 poziții libere. Numerotînd placa liberă prin 0, putem scrie:

```
type Stare = array [1..4, 1..4] of 0..15;
```

Menționăm că numărul de stări posibile este dat de numărul aranjamentelor 16 din 16 și constituie:

$$A_{16}^{16} = P_{16} = 16! \approx 2,09 \cdot 10^{13},$$

pe cînd capacitatea memoriei interne a calculatoarelor personale este de ordinul 10^8 octeți.

Trecerea dintr-o stare în alta poate fi efectuată prin „deplasarea” plăcuței lipsă spre stînga, sus, dreapta, jos (bineînțeles cînd acest lucru este posibil). În PASCAL această „deplasare” se exprimă prin modificarea componentelor respective ale variabilelor de tip Stare.

Cunoscînd stările sistemului și regulile de trecere dintr-o stare în alta, putem construi **arborele soluțiilor**, o parte din care este prezentată în figura 5.13.

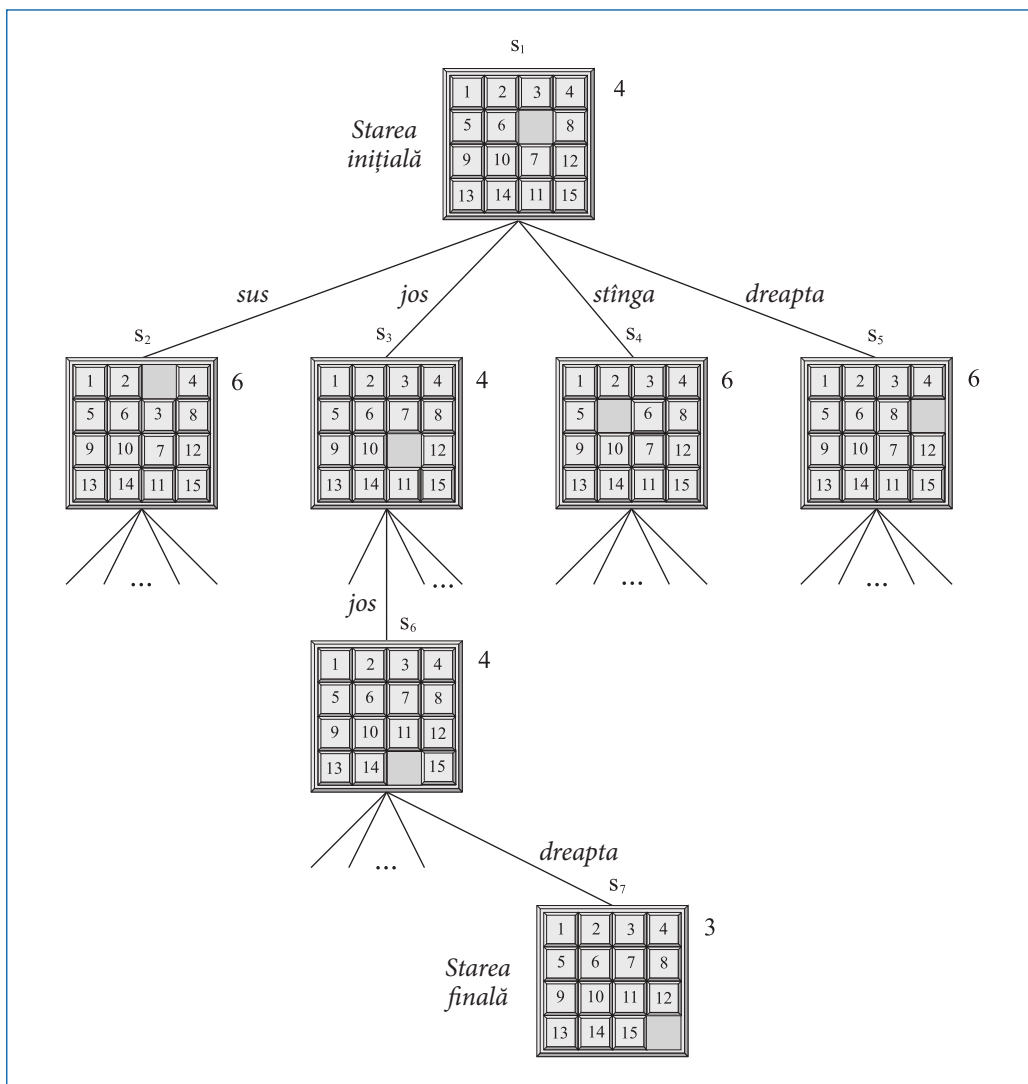


Fig. 5.13. Arborele soluțiilor în jocul Perspico

Întrucît plăcuța liberă poate fi deplasată pe trasee ciclice, arborele soluțiilor este infinit. Pentru a evita revenirea la stările deja examinate, vom include în arbore numai stările curente noi. De asemenea, luînd în considerare faptul că memoria internă a calculatoarelor personale nu permite stocarea tuturor stărilor posibile, în continuare vom examina numai primele 10–15 nivele ale arborelui soluțiilor. Evident, în prezența acestei restricții nu se mai garantează găsirea, chiar dacă există, a șirului de mutări permise care transformă starea inițială în stare finală.

Funcția de cost $f: S \rightarrow R$ poate fi definită în forma:

$$f(s_i) = niv(s_i) + h(s_i),$$

unde componenta $h(s_i)$ caracterizează numărul de mutări necesare pentru a trece din starea curentă s_i în stare finală. Întrucît acest număr nu poate fi calculat fără a

construi mai întâi subarboarele de rădăcină s_i , vom folosi o aproximare prin lipsă a numărului necesar de mutări:

$$h(s_i) = \text{numărul de plăcuțe care nu se află la locul lor.}$$

De exemplu, pentru arborele din figura 5.13 avem:

$$f(s_1) = \text{niv}(s_1) + h(s_1) = 0 + 4 = 4;$$

$$f(s_2) = \text{niv}(s_2) + h(s_2) = 1 + 5 = 6;$$

$$f(s_3) = \text{niv}(s_3) + h(s_3) = 1 + 3 = 4;$$

$$f(s_6) = \text{niv}(s_6) + h(s_6) = 2 + 2 = 4;$$

$$f(s_7) = \text{niv}(s_7) + h(s_7) = 3 + 0 = 3.$$

Arborele soluțiilor și lista nodurilor active pot fi reprezentate în memoria calculatorului cu ajutorul următoarelor structuri de date:

```

type AdresaNod = ^Nod;
  Nod = record
    S : Stare;
    Nivel, Cost : integer;
    Drum : boolean;
    D : array [1..4] of AdresaNod;
    Tata : AdresaNod;
  end;
  AdresaCelula = ^Celula;
  Celula = record
    ReferintaNod : AdresaNod;
    Urm : AdresaCelula;
  end;
var Radacina : AdresaNod;
    BazaListei : AdresaCelula;

```

Fiecare nod al arborelui soluțiilor conține câmpurile informaționale S, Nivel, Cost, Drum și câmpurile de legătură D, Tata. Câmpul Drum se folosește pentru a marca nodurile ce aparțin drumului care leagă nodul rădăcină cu nodul ce conține starea finală. Câmpul Tata se utilizează pentru a memora adresa nodului tată.

Fiecare celulă a listei nodurilor active conține câmpul informațional ReferintaNod în care se indică adresa nodului inclus în listă. Câmpul de legătură Urm conține adresa celei următoare din listă. Prin urmare, lista nodurilor active conține nu nodurile propriu-zise, ci doar adresele lor.

Subprogramele necesare pentru implementarea metodei ramifică și mărginește în cazul jocului Perspico sînt incluse în programul P158.

```

Program P158;
  { Jocul Perspico - metoda ramifica si margineste }
  const NivelMaxim=15;
  type Stare=array[1..4, 1..4] of 0..15;
    AdresaNod=^Nod;

```

```

    Nod=record
        S : Stare;
        Nivel, Cost : integer;
        Drum : boolean;
        D : array [1..4] of AdresaNod; {adresele
                                         descendente}

        Tata : AdresaNod;
    end;
AdresaCelula=^Celula;
Celula=record
    ReferintaNod : AdresaNod;
    Urm : AdresaCelula;
end;

var Radacina : AdresaNod;
    BazaListei : AdresaCelula;
    StareaInitiala, StareaFinala : Stare;
    Gasit : boolean; { true cind este gasit un drum }
    Finput, Foutput : text; { datele de intrare si iesire }
    Str : array[1..4] of Stare; { starile descendente }
    m : 0..4; { numarul curent de stari }
    i, j : integer;

procedure CalculareaCostului(Adresa : AdresaNod);
var i, j, C : integer;
begin
    C:=0;
    for i:=1 to 4 do
        for j:=1 to 4 do
            if Adresa^.S[i,j] <> StareaFinala[i,j] then C:=C+1;
        Adresa^.Cost:=Adresa^.Nivel+C;
    end; { CalculareaCostului }

procedure Initializare;
{ Creeaza nodul radacina si lista nodurilor active }
{ Inscribe in lista nodurilor active nodul radacina }
var i : integer;
begin
    Gasit:=false;
    new(Radacina);
    Radacina^.S:=StareaInitiala;
    Radacina^.Nivel:=0;
    CalculareaCostului(Radacina);
    Radacina^.Drum:=false;
    for i:=1 to 4 do Radacina^.D[i]:=nil;
    Radacina^.Tata:=nil;

```

```

    new(BazaListei);
    BazaListei^.ReferintaNod:=Radacina;
    BazaListei^.Urm:=nil;
end; { Initializare }

procedure Ramifica(Adresa : AdresaNod);
    { Inscribe in Str toate starile care pot fi obtinute }
    { din starea curenta prin efectuarea unei mutari permise }
label l;
var    St : Stare;
        i, j, k : integer;
begin
    { cautarea placutei 0 }
    for i:=1 to 4 do
        for j:=1 to 4 do
            if Adresa^.S[i,j]=0 then goto l;
l: m:=0;
    { Deplasarea placutei de sus }
    if i<>1 then
        begin
            St:=Adresa^.S;
            St[i,j]:=St[i-1, j];
            St[i-1, j]:=0;
            m:=m+1;
            Str[m]:=St;
        end;
    { Deplasarea placutei de jos }
    if i<>4 then
        begin
            St:=Adresa^.S;
            St[i,j]:=St[i+1, j];
            St[i+1, j]:=0;
            m:=m+1;
            Str[m]:=St;
        end;
    { Deplasarea placutei din stinga }
    if j<>1 then
        begin
            St:=Adresa^.S;
            St[i,j]:=St[i, j-1];
            St[i, j-1]:=0;
            m:=m+1;
            Str[m]:=St;
        end;
    { Deplasarea placutei din dreapta }
    if j<>4 then

```



```

begin
    St:=Adresa^.S;
    St[i,j]:=St[i, j+1];
    St[i, j+1]:=0;
    m:=m+1;
    Str[m]:=St;
end;
end; { Desparte }

procedure IncludeInLista(Adresa : AdresaNod);
{ Include adresa nodului in lista nodurilor active }
var R : AdresaCelula;
begin
    new(R);
    R^.ReferintaNod:=Adresa;
    R^.Urm:=BazaListei;
    BazaListei:=R;
end; { IncludeInLista }

procedure ExtrageDinLista(var Adresa : AdresaNod);
{ Extrage adresa nodului de cost minim din lista }
label 1;
var P, R : AdresaCelula;
    C : integer; { costul curent }
begin
    if BazaListei=nil then goto 1;
    { cautarea nodului de cost minim }
    C:=MaxInt;
    R:=BazaListei;
    while R<>nil do
        begin
            if R^.ReferintaNod^.Cost < C then
                begin
                    C:=R^.ReferintaNod^.Cost;
                    Adresa:=R^.ReferintaNod;
                    P:=R;
                end; { then }
            R:=R^.Urm;
        end; { while }
    { excludera nodului de cost minim din lista }
    if P=BazaListei then BazaListei:=P^.Urm
    else
        begin
            R:=BazaListei;
            while P<>R^.Urm do R:=R^.Urm;
            R^.Urm:=P^.Urm;
        end;
end;

```

```

    dispose(P);
1:end; { ExtrageDinLista }

function StariEgale(S1, S2 : Stare) : boolean;
    { Returneaza TRUE daca starile S1 si S2 coincid }
var i, j : integer;
    Coincid : boolean;
begin
    Coincid:=true;
    for i:=1 to 4 do
        for j:=1 to 4 do
            if S1[i,j]<>S2[i,j] then Coincid:=false;
    StariEgale:=Coincid;
end; { StariEgale }

function StareDejaExaminata(St : Stare) : boolean;
    { Returneaza TRUE daca starea curenta este deja inclusa in
    arbore }
var EsteInArbore : boolean;

procedure InAdincime(R : AdresaNod);
    { Parcurgerea arborelui in adincime }
label 1;
var i : integer;
begin
    if R<>nil then
        begin
            if StariEgale(St, R^.S) then
                begin
                    EsteInArbore:=true;
                    goto 1;
                end;
            for i:=1 to 4 do InAdincime(R^.D[i]);
        end;
1:end; { InAdincime }

begin
    EsteInArbore:=false;
    InAdincime(Radacina);
    StareDejaExaminata:=EsteInArbore;
end; { StareDejaExaminata }

procedure Margineste(Adresa : AdresaNod);
    { Selectarea descendentilor si includerea lor in lista }
label 1;
var i, k : integer;
    R : AdresaNod;

```

```

begin
  k:=0;
  if (Adresa^.Nivel+1) > NivelMaxim then goto l;
  for i:=1 to m do
    if not StareDejaExaminata(Str[i]) then
      begin
        k:=k+1;
        new(R);
        R^.S:=Str[i];
        R^.Nivel:=Adresa^.Nivel+1;
        CalculareaCostului(R);
        for j:=1 to 4 do R^.D[j]:=nil;
        Adresa^.D[i]:=R;
        R^.Tata:=Adresa;
        R^.Drum:=false;
        if StariEgale(R^.S, StareaFinala) then
          begin
            R^.Drum:=true;
            Gasit:=true;
          end;
        IncludeInLista(R);
      end;
    writeln(Foutput);
    writeln(Foutput, 'In lista au fost inclusi ', k, '
descendenti');
    writeln(Foutput);
  l:end; { Margineste }

procedure AfisareaNodului(R : AdresaNod);
var i, j : integer;
begin
  writeln(Foutput, 'Drum=', R^.Drum, ' Nivel=', R^.Nivel,
          ' Cost=', R^.Cost);
  for i:=1 to 4 do
    begin
      for j:=1 to 4 do write(Foutput, R^.S[i, j] : 3);
      writeln(Foutput);
    end;
    for i:=1 to 4 do
      if R^.D[i]<>nil then write(Foutput, '*** ');
      else write(Foutput, 'nil ');
    writeln(Foutput); writeln(Foutput);
  end; { AfisareaNodului }

procedure RamificaSiMargineste;
var NodulCurent : AdresaNod;

```

```

begin
  Initializare;
  repeat
    ExtrageDinLista(NodulCurent);
    writeln(Foutput, '    NODUL EXTRAS DIN LISTA');
    writeln(Foutput, '    =====');
    AfisareaNodului(NodulCurent);
    Ramifica(NodulCurent);
    Margineste(NodulCurent);
  until Gasit or (BazaListei=nil);
end; { RamificaSiMargineste }

procedure AfisareaDrumului;
label 1;
var R : AdresaCelula;
    P, Q : AdresaNod;
begin
  if not Gasit then
    begin
      writeln(Foutput, 'DRUMUL NU A FOST GASIT');
      goto 1;
    end;
    writeln(Foutput, '    DRUMUL GASIT:');
    writeln(Foutput, '    =====');
    { cautarea in lista a nodului terminal}
    R:=BazaListei;
    while (R<>nil) and (not R^.ReferintaNod^.Drum) do R:=R^.Urm;
    { marcarea nodurilor care formeaza drumul }
    P:=R^.ReferintaNod;
    while P<>nil do
      begin
        P^.Drum:=true;
        P:=P^.Tata;
      end;
    { afisarea drumului }
    P:=Radacina;
    while P<>nil do
      begin
        AfisareaNodului(P);
        Q:=nil;
        for i:=1 to 4 do
          if (P^.D[i]<>nil) and P^.D[i]^^.Drum then Q:=P^.D[i];
        P:=Q;
      end;
    writeln(Foutput, 'Sfirsitul drumului');
  1:end; { AfisareaDrumului }

```

```

begin
  { Citirea starii initiale }
  assign(Finput, 'FINPUT.TXT');
  reset(Finput);
  for i:=1 to 4 do
    for j:=1 to 4 do read(Finput, StareaInitiala[i, j]);
  { Citirea starii finale }
  for i:=1 to 4 do
    for j:=1 to 4 do read(Finput, StareaFinala[i, j]);
  close(Finput);
  { Deschiderea fisierului de iesire }
  assign(Foutput, 'FOUTPUT.TXT');
  rewrite(Foutput);
  RamificaSiMargineste;
  AfisareaDrumului;
  close(Foutput);
  writeln('Gasit=', Gasit);
  readln;
end.

```

În programul P158 starea inițială și starea finală sînt citite din fișierul FINPUT.TXT, iar nodurile extrase la fiecare pas din listă și drumul găsit sînt înscrise în fișierul FOUTPUT.TXT. Conținutul acestor fișiere poate fi vizualizat sau modificat cu ajutorul unor editoare simple de text.

Întrebări și exerciții

- ❶ Indicați ordinea în care vor fi vizitate nodurile arborelui din *figura 5.13* în cazul parcurgerii de cost minim. Folosind ca model *tabelul 5.2*, completați un tabel cu datele referitoare la acest arbore.
- ❷ Explicați destinația fiecărui subprogram din programul P158.
- ❸ Lansați în execuție programul P158 pentru datele inițiale din *figura 5.12*. Comentați rezultatele înscrise în fișierul de ieșire.
- ❹ Lansați în execuție programul P158 pentru următoarele stări inițiale ale jocului Perspico:

a)

1	2	0	4
5	6	3	8
9	11	7	12
13	10	14	15

b)

1	2	3	4
5	7	6	8
9	0	10	11
13	14	15	12

c)

1	3	4	8
5	2	6	0
9	10	7	11
13	14	15	12

d)

0	1	2	3
6	7	8	4
5	9	10	11
13	14	15	12

Comentați rezultatele înscrise de program în fișierul FOUTPUT.TXT.

5 E cunoscut faptul că pentru stările inițiale ce urmează:

a)

1	3	4	4
9	5	7	8
13	6	10	11
0	14	15	12

b)

1	2	3	4
5	0	7	8
9	6	10	11
13	14	15	12

programul P158 găsește soluțiile respective. Însă, după introducerea modificării `const NivelMaxim = 5;`

soluția pentru cazul (a) nu mai este găsită. Cum credeți, prin ce se explică acest fapt?

6 Se consideră varianta simplificată a jocului Perspico, denumită **jocul 9** (fig. 5.14). Ce modificări trebuie introduse în programul P158 pentru a calcula șirul de mutări permise destinate trecerii din starea inițială în starea finală?

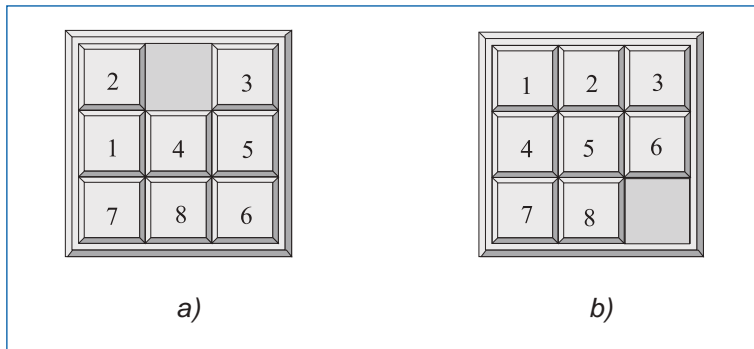


Fig. 5.14. Varianta simplificată a jocului Perspico:
a – starea inițială; b – starea finală

7 Modificați programul P158 utilizând următoarea funcție de cost:

$$F(s_i) = niv(s_i) + g(s_i),$$

unde $g(s_i)$ este suma distanțelor dintre fiecare plăcuță și locul ei în starea finală, exprimată prin numărul respectiv de mutări. De exemplu, pentru starea inițială din figura 5.12 avem:

$$g(s_i) = 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 3 = 6.$$

Verificați cum derulează programul modificat în cazul stărilor inițiale din *exercițiile 4 și 5*. Elaborați planul unui experiment care ne-ar permite să stabilim care din funcțiile $h(s)$, $g(s)$ micșorează timpul de calcul.

- ⑧ Comparați complexitatea programului P158 cu complexitatea unui program bazat pe metoda trierii.
- ⑨ Se consideră o tablă de șah pe care se află o singură piesă – un cal de culoare albă. Elaborați un program care determină dacă piesa respectivă poate trece, utilizând mutările permise, din poziția (α, β) în poziția (γ, δ) . Amintim că $\alpha, \gamma \in \{A, B, \dots, H\}$, iar $\beta, \delta \in \{1, 2, \dots, 8\}$.
- ⑩ **Problema comis-voiajorului***. Se consideră n orașe între care există curse aeriene directe. În orașul i se află un comis-voiajor care trebuie să viziteze celelalte $n-1$ orașe și să revină în localitatea din care s-a pornit. Elaborați un program care, cunoscând distanțele d_{ij} între orașe, determină ordinea în care ele vor fi vizitate. Se cere ca distanța parcursă de comis-voiajor să fie cât mai mică, iar fiecare oraș să fie vizitat numai o singură dată.
- ⑪ Scrieți un program PASCAL care soluționează problema comis-voiajorului prin metoda trierii. Comparați complexitatea programului elaborat cu complexitatea programului bazat pe metoda *ramifică și mărginește*.
Indicații: Drumul parcurs de comis-voiajor poate fi reprezentat ca o mulțime ordonată $(1, i, j, \dots, k, 1)$ de orașe în care $i, j, \dots, k \in \{2, 3, \dots, n\}$. Pentru a genera toate drumurile posibile, se calculează permutările mulțimii $\{2, 3, \dots, n\}$.
- ⑫ Efectuați un studiu comparativ al algoritmilor bazați pe metoda *ramifică și mărginește* și algoritmilor bazați pe metoda trierii.

5.9. Algoritmi exacți și algoritmi euristici

Algoritmii utilizați pentru rezolvarea problemelor cu ajutorul calculatorului se împart în două categorii distincte: algoritmi exacți și algoritmi euristici.

Un algoritm este **exact** numai atunci când el găsește soluțiile optime ale problemelor pentru rezolvarea cărora a fost conceput. Desigur, acest fapt trebuie confirmat printr-o demonstrație matematică riguroasă.

Un algoritm este **euristic** atunci când el găsește soluții bune, dar nu neapărat optime. În astfel de cazuri demonstrația matematică respectivă nu există sau nu este cunoscută. De obicei, algoritmii euristici includ prelucrări care, chiar dacă nu pot fi justificate matematic, sînt alese în virtutea experienței sau intuiției programatorului.

Evident, algoritmii bazați pe metoda trierii sînt exacți, întrucît în procesul examinării consecutive a soluțiilor posibile neapărat va fi găsită și soluția optimă. Confirmarea exactității acestor algoritmi se reduce la demonstrarea faptului că în procesul derulării pe calculator vor fi generate și analizate toate elementele din mulțimea soluțiilor posibile. În cazul algoritmilor bazați pe celelalte tehnici de programare – tehnica *Greedy*, metoda reluării, metoda ramifică și mărginește etc. – un algoritm va fi exact sau euristic în funcție de natura condițiilor care ne permit să evităm

* Persoană care se ocupă cu mijlocirea vînzărilor de mărfuri, deplasîndu-se în diferite locuri în căutarea unor beneficiari.

trierea tuturor soluțiilor posibile. Dacă aceste condiții sînt alese nereușit, soluțiile optime pot fi pierdute și, în consecință, algoritmul respectiv nu va mai fi exact. Pentru exemplificare vom analiza problema ce urmează.

Problema drumului minim. Se consideră n orașe legate printr-o rețea de drumuri (fig. 5.15). Cunoscînd distanțele d_{ij} dintre orașele vecine, determinați cel mai scurt drum din orașul a în orașul b .

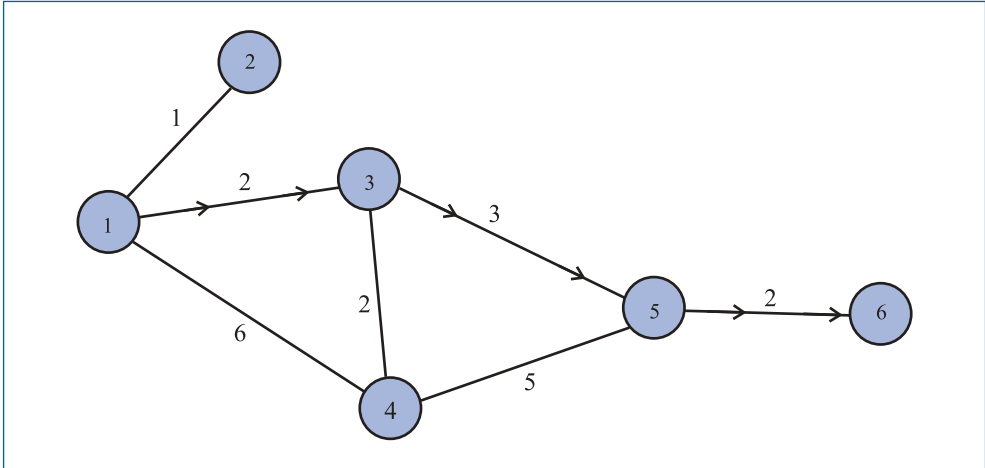


Fig. 5.15. Rețea de drumuri

Datele inițiale ale problemei în studiu pot fi descrise cu ajutorul matricei (tabelului bidimensional) $D = \|d_{ij}\|_{n \times n}$ cu n linii și n coloane, denumită **matricea distanțelor**. În această matrice componenta d_{ij} este egală cu distanța dintre orașele i, j atunci cînd ele sînt vecine și 0 în caz contrar. Prin definiție, $d_{ii} = 0, i = 1, 2, \dots, n$.

De exemplu, pentru orașele din figura 5.15 avem:

$$D = \begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 2 & 6 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & 0 & 2 & 3 & 0 \\ 4 & 6 & 0 & 2 & 0 & 5 & 0 \\ 5 & 0 & 0 & 3 & 5 & 0 & 2 \\ 6 & 0 & 0 & 0 & 0 & 2 & 0 \end{array}$$

Drumul minim ce leagă orașele $a = 1, b = 6$ are lungimea 7 și include localitățile 1, 3, 5, 6.

În general, drumul minim poate fi găsit prin **metoda trierii**, generînd consecutiv toate permutările mulțimilor ordonate

$$X = (a, \underbrace{i, j, \dots, k}_q \text{ orașe}, b),$$

unde q ia valori de la 0 la $n-2$. Este clar că algoritmi bazați pe generarea tuturor permutărilor întotdeauna determină drumul minim, însă complexitatea temporală $O(n!)$ a acestor algoritmi este inacceptabilă.

Pentru a reduce volumul de calcule, vom încerca să determinăm drumul minim prin **metoda reluării**. În această metodă construirea drumului începe cu orașul inițial $x_1 = a$ și continuă cu primul dintre vecinii săi nevizitați, fie acesta x_2 , trecându-se la primul dintre vecinii lui x_2 nevizitați încă ș.a.m.d. Pentru a construi drumuri cât mai scurte, vom aplica următoarea **regulă intuitivă**: la fiecare pas vom examina, în primul rând, vecinii nevizitați care se află cât mai aproape de orașul curent.

Drumul în construcție poate fi reprezentat în forma unui vector:

$$X = (a, x_2, \dots, x_{k-1}, x_k, \dots, b),$$

în care componenta x_k trebuie să fie unul din vecinii orașului x_{k-1} . Pentru a sistematiza calculele vom memora vecinii încă nevizitați ai orașului i în mulțimea A_i , $i = 1, 2, \dots, n$. Evident, conform regulii intuitive formulate mai sus, orașele din fiecare mulțime A_i trebuie sortate în ordinea creșterii distanțelor d_{ij} , $j \in A_i$.

De exemplu, pentru *figura 5.15* inițial vom avea:

$$A_1 = (2, 3, 4);$$

$$A_2 = (1);$$

$$A_3 = (1, 4, 5);$$

$$A_4 = (3, 5, 6);$$

$$A_5 = (6, 3, 4);$$

$$A_6 = (5).$$

Condițiile de continuare în metoda reluării rezultă direct din enunțul problemei: orașul x_k poate fi adăugat la porțiunea de drum deja construită (a, x_2, \dots, x_{k-1}) numai atunci când:

1) x_k este un vecin încă nevizitat al orașului x_{k-1} , deci $x_k \in A_{k-1}$;

2) orașul x_k anterior nu a fost inclus în drumul în curs de construcție, deci $x_k \neq a$, $x_k \neq x_2, \dots, x_k \neq x_{k-1}$.

De exemplu, pentru *figura 5.15* vectorul X va lua consecutiv următoarele valori:

$$X = (1);$$

$$X = (1, 2);$$

$$X = (1);$$

$$X = (1, 3);$$

$$X = (1, 3, 4);$$

$$X = (1, 3, 4, 5);$$

$$X = (1, 3, 4, 5, 6).$$

Drumul $(1, 3, 4, 5, 6)$ construit prin metoda reluării are lungimea 11 și, evident, nu este un drum minim. Totuși acest drum este mai bun ca drumul $(1, 4, 5, 6)$ care are lungimea 13.

În programul ce urmează elementele mulțimilor A_1, A_2, \dots, A_n sînt plasate la începutul liniilor $A[1], A[2], \dots, A[n]$ ale tabelului bidimensional A , restul pozițiilor avînd valoarea zero.

```

Program P159;
  { Problema drumului minim - metoda reluării }
const nmax=50;
var n : integer; { numărul de orașe }
    D : array[1..nmax, 1..nmax] of real; { matricea distanțelor }
    a, b : 1..nmax;
    X : array [1..nmax] of integer;      { drumul construit }
    V : array[1..nmax, 1..nmax] of integer; { vecinii }
    Finput : text;

procedure InitializareVecini;
  { înscrie în V[k] vecinii orașului k }
var k, i, j, p, q, r : integer;
begin
  for k:=1 to n do
    begin
      { inițial mulțimea V[k] este vidă }
      for i:=1 to n do V[k,i]:=0;
      { calculăm elementele mulțimii V[k] }
      i:=0;
      for j:=1 to n do
        if D[k,j]<>0 then
          begin
            i:=i+1;
            V[k,i]:=j;
          end; { then }
      { sortarea elementelor mulțimii V[k] prin metoda bulelor }
      for j:=1 to i do
        for p:=1 to i-1 do
          if D[k, V[k,p]]>D[k, V[k, p+1]] then
            begin
              q:=V[k,p];
              V[k,p]:=V[k, p+1];
              V[k, p+1]:=q;
            end; { then }
        end; { for }
    end; { InitializareVecini }

procedure Initializare;
var i, j : integer;
begin
  assign(Finput, 'DRUM.IN');
  reset(Finput);
  readln(Finput, n);
  readln(Finput, a, b);
  writeln('n=', n, ' a=', a, ' b=', b);
  for i:=1 to n do
    for j:=1 to n do read(Finput, D[i,j]);

```

```

close(Finput);
InitializareVecini;
end; { Initializare }

function MultimeVida(k : integer) : boolean;
begin
    MultimeVida:=(V[X[k-1], 1]=0);
end; { MultimeVida }

function PrimulElement(k : integer) : integer;
var i : integer;
begin
    PrimulElement:=V[X[k-1], 1];
    for i:=1 to n-1 do V[X[k-1],i]:=V[X[k-1], i+1];
end; { PrimulElement }

function ExistaSuccesor(k : integer) : boolean;
begin
    ExistaSuccesor:=(V[X[k-1], 1]<>0);
end; { ExistaSuccesor }

function Succesor(k : Integer) : integer;
var i : integer;
begin
    Succesor:=V[X[k-1], 1];
    for i:=1 to n-1 do V[X[k-1], i]:=V[X[k-1], i+1];
end; { Succesor }

function Continuare(k : integer) : boolean;
var i : integer;
Indicator : boolean;
begin
    Continuare:=true;
    for i:=1 to k-1 do
        if X[i]=X[k] then Continuare:=false;
    end; { Continuare }

procedure PrelucrareaSolutiei(k : integer);
var i : integer;
    s : real;
begin
    writeln('Drumul gasit:');
    for i:=1 to k-1 do write(X[i] : 3);
writeln;
    s:=0;
    for i:=1 to k-1 do s:=s+D[X[i], X[i+1]];
    writeln('Lungimea drumului ', s : 10:2);

```

```

    readln;
    halt;
end; { PrelucrareaSolutiei }

procedure Reluare(k : integer);
label l;
var i : integer;
begin
    if MultimeVida(k) then goto l;
    if X[k-1]<>b then
        begin
            X[k]:=PrimulElement(k);
            if Continuare(k) then Reluare(k+1);
            while ExistaSuccesor(k) do
                begin
                    X[k]:=Succesor(k);
                    if Continuare(k) then Reluare(k+1);
                end; { while }
            end { then}
        else PrelucrareaSolutiei(k);
    l:end; { Reluare }

begin
    Initializare;
    X[1]:=a;
    Reluare(2);
end.

```

Din analiza programului P159 se observă că algoritmul bazat pe metoda reluării are complexitatea $O(m^n)$, deci este exponențial. În această formulă m reprezintă numărul maxim de vecini pe care îi poate avea fiecare oraș. Menționăm că numărul concret de operații efectuate de programul P159 depinde în mare măsură de configurația rețelei de drumuri și distanțele respective. Evident, „plata” pentru reducerea numărului de operații în algoritmul euristic bazat pe metoda reluării în comparație cu algoritmul exact bazat pe metoda trierii constă în pierderea soluțiilor optime.

În scopul soluționării eficiente a problemelor de o reală importanță practică, în informatică se depun eforturi considerabile pentru elaborarea unor algoritmi exacți care ar avea o complexitate polinomială. În cazul problemei drumului minim un astfel de algoritm poate fi elaborat utilizând metoda **programării dinamice**.

Amintim că metoda programării dinamice poate fi aplicată doar problemelor care satisfac principiul optimalității, principiul care se respectă în cazul drumurilor minime. Pentru a formula relațiile respective de recurență, introducem în studiu **matricea costurilor** $C = \|c_{ij}\|_{n \times n}$, elementele căreia indică lungimea drumului minim între orașele i, j . Componentele c_{ij} se calculează după cum urmează:

1) mai întâi se examinează drumurile minime care leagă orașele vecine:

$$c_{ij} = \begin{cases} 0, & \text{dacă } i = j; \\ d_{ij}, & \text{dacă orașele } i, j \text{ sînt vecine;} \\ \infty, & \text{în caz contrar;} \end{cases}$$

2) în continuare se examinează drumurile minime între orașele i, j formate prin folosirea localității k drept punct de conexiune:

$$c_{ij} = \min(c_{ik}, c_{ik} + c_{jk}), \quad i, j \in \{1, 2, \dots, n\}, \quad i \neq j, \quad i \neq k, \quad j \neq k;$$

3) punctul 2 se repetă pentru $k = 1, 2, \dots, n$.

Valorile curente ale matricei costurilor pentru orașele din figura 5.15 sînt prezentate în figura 5.16.

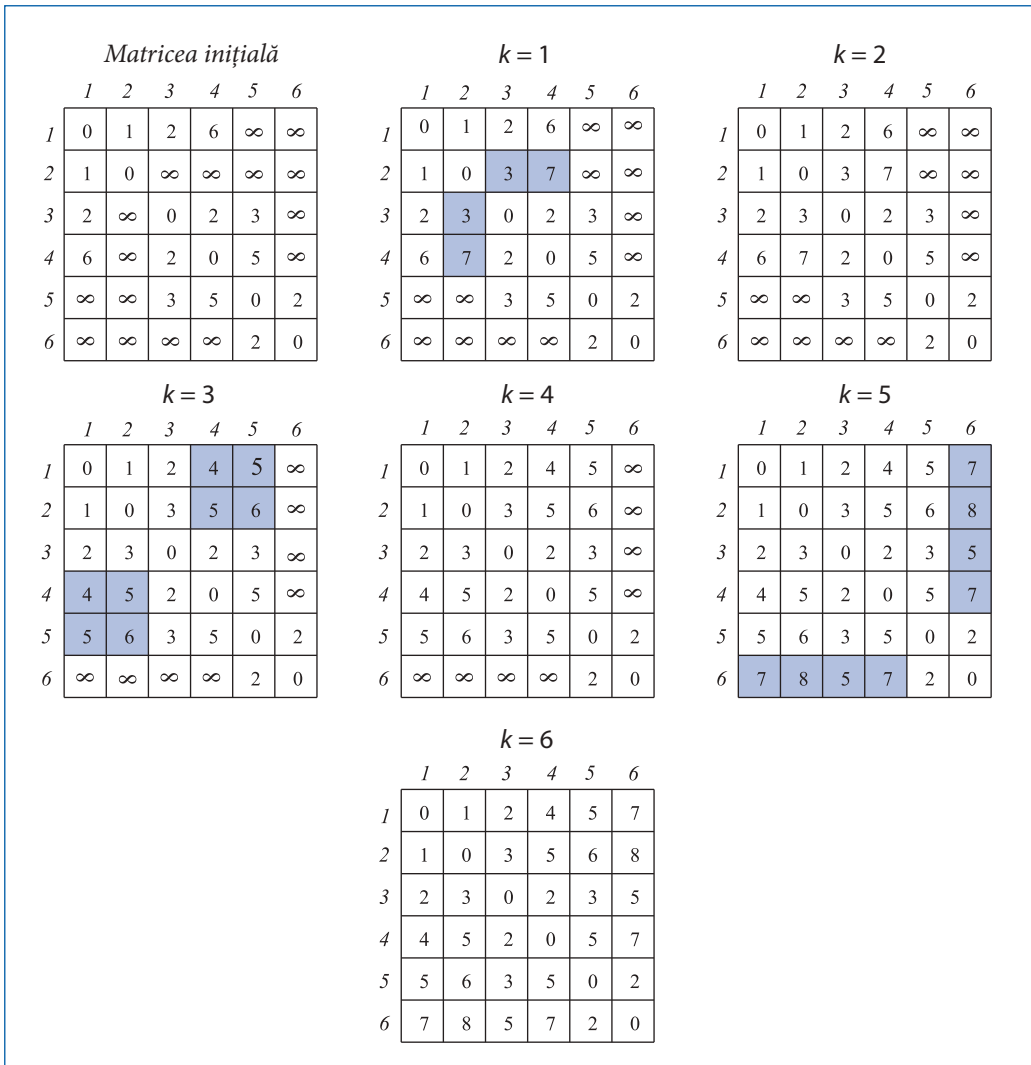


Fig. 5.16. Valorile curente ale matricei costurilor

Întrucît matricea costurilor C nu include drumurile minime propriu-zise, ci numai lungimile lor, vom construi drumul minim ce leagă orașele a, b cu ajutorul tehnicii *Greedy*. Conform acestei tehnici, construcția drumului minim $X = (x_1, \dots, x_{k-1}, x_k, \dots)$ începe cu orașul inițial $x_1 = a$. În continuare, la fiecare pas k se alege acel oraș $x_k, x_k \in A_{k-1}$, care satisface principiul optimalității:

$$C[a, x_k] + C[x_k, b] = C[a, b].$$

De exemplu, pentru orașele $a = 1, b = 6$ din figura 5.15 avem:

$$\begin{aligned} X &= (1); \\ X &= (1, 3); \\ X &= (1, 3, 5); \\ X &= (1, 3, 5, 6). \end{aligned}$$

Algoritmul de determinare a drumurilor minime bazat pe metoda programării dinamice este cunoscut în literatura de specialitate sub denumirea *Roy-Floyd*. În calitate de exercițiu vă propunem să demonstrați că acest algoritm este exact, adică întotdeauna construiește numai drumuri minime.

În programul ce urmează algoritmul exact este implementat cu ajutorul procedurii RoyFloyd.

```

Program P160;
  { Problema drumului minim - metoda programării dinamice }
const nmax=50;
         Infinit=1.0E+35;
var n : integer; { numarul de orase }
      D : array[1..nmax, 1..nmax] of real; { matricea distanțelor }
      a, b : 1..nmax;
      X : array [0..nmax+1] of integer;      { drumul construit }
      V : array[1..nmax, 1..nmax] of integer; { vecinii }
      C : array[1..nmax, 1..nmax] of real; { matricea costurilor }
      Finput : text;

procedure InitializareVecini;
  { înscrie în V[k] vecinii orașului k }
var k, i, j, p, q, r : integer;
begin
  for k:=1 to n do
    begin
      { inițial multimea V[k] este vidă }
      for i:=1 to n do V[k,i]:=0;
      { calculăm elementele mulțimii V[k] }
      i:=0;
      for j:=1 to n do
        if D[k,j]<>0 then
          begin
            i:=i+1;
            V[k,i]:=j;
          end
        end
    end

```

```

        end; { then }
    end; { for }
end; { InitializareVecini }

procedure Initializare;
var i, j : integer;
begin
    assign(Finput, 'DRUM.IN');
    reset(Finput);
    readln(Finput, n);
    readln(Finput, a, b);
    writeln('n=', n, ' a=', a, ' b=', b);
    for i:=1 to n do
        for j:=1 to n do read(Finput, D[i,j]);
    close(Finput);
    InitializareVecini;
end; { Initializare }

procedure AfisareaDrumului;
var k : integer;
begin
    write('Drumul găsit: ');
    k:=1;
    repeat
        write(X[k] : 3);
        k:=k+1;
    until X[k]=0;
    writeln;
    writeln('Lungimea drumului ', C[a, b] : 5);
    readln;
end; { PrelucrareaSolutiei }

function Min(p, q : real) : real;
    { Returnează minimul din p si q }
var s : real;
begin
    if p<q then s:=p else s:=q;
    if s>Infinit then s:=Infinit;
    Min:=s;
end; { Min }

procedure RoyFloyd;
var i, j, k : integer;
    s : real;
    ors : integer; { orașul candidat la includerea în drumul
                    minim }
    cnd : boolean; { condițiile de includere a orașului în drum }

```

```

begin
  { Inițializarea matricei costurilor }
  for i:=1 to n do
    for j:=1 to n do
      if (D[i,j]=0) and (i<>j) then C[i,j]:=Infinit
      else C[i,j]:=D[i,j];
    { Calcularea matricei costurilor }
  for k:=1 to n do
    for i:=1 to n do
      if i<>k then
        for j:=1 to n do
          if j<>k then C[i,j]:=Min(C[i,j], C[i,k]+C[j,k]);
        { Trasarea drumului - tehnica Greedy }
    for k:=1 to n do X[k]:=0;
  k:=1; X[1]:=a;
  while X[k]<>b do
    begin
      i:=1;
      while V[X[k], i]<>0 do
        begin
          ors:=V[X[k], i];
          cnd:=true;
          for j:=1 to k do if ors=X[j] then cnd:=false;
          if cnd and (C[A, ors]+C[ors, B]=C[a,b])
            then X[k+1]:=ors;
          i:=i+1;
        end; { while }
      k:=k+1
    end; { while }
  end; { RoyFloyd }

begin
  Initializare;
  RoyFloyd;
  AfisareaDrumului;
end.

```

Din analiza procedurii RoyFloyd se observă că complexitatea temporală a algoritmului exact este $O(n^3)$, deci polinomială.

În practică, pentru soluționarea unei probleme, mai întâi se încearcă elaborarea unui algoritm exact de complexitate polinomială. Dacă această tentativă eșuează, se elaborează un algoritm euristic. Pentru a sistematiza acest proces este convenabil să se pună în evidență toate condițiile pe care le satisface o soluție optimă. Condițiile respective pot fi împărțite în două clase:

1) condiții necesare în sensul că neîndeplinirea lor împiedică obținerea unei soluții posibile pentru problemă;

2) condiții pentru care se poate accepta un compromis, în sensul că ele pot fi înlocuite cu alte condiții ce permit apropierea de o soluție optimală.

De exemplu, în cazul drumului minim $X = (a, x_2, \dots, x_{k-1}, x_k, \dots, b)$ faptul că x_k trebuie să fie un vecin al orașului x_{k-1} este o condiție necesară, iar respectarea principiului optimalității este o condiție de compromis. În metoda reluării condiția de compromis a fost înlocuită cu una mai simplă, și anume, orașul x_k trebuie să fie cât mai aproape de orașul x_{k-1} . Evident, formularea condițiilor care acceptă un compromis și înlocuirea lor cu altele mai simple cade în sarcina programatorului.

Întrebări și exerciții

- ❶ Care este diferența dintre algoritmii exacti și cei euristici?
- ❷ În care cazuri algoritmii euristici sînt „mai buni” decît cei exacti?
- ❸ Scrieți un program care determină cel mai scurt drum prin metoda trierii. Estimați complexitatea temporală a programului elaborat.
- ❹ Efectuați un studiu comparativ al algoritmilor exacti și algoritmilor euristici destinați soluționării problemei drumului minim.
- ❺ Calculați matricea distanțelor pentru rețeaua de drumuri auto ce leagă centrele raionale. Determinați cele mai scurte drumuri între centrele raionale date cu ajutorul algoritmilor exacti și algoritmilor euristici.
- ❻ Cum credeți, poate fi oare aplicată metoda trierii pentru determinarea celui mai scurt drum între oricare două localități din țară? Argumentați răspunsul dvs.
- ❼ Formulați condițiile necesare și condițiile pentru care se poate accepta un compromis în cazul robotului ce explorează un teren aurifer (vezi paragraful 5.6).
- ❽ Estimați complexitatea algoritmilor exacti și a algoritmilor euristici destinați soluționării problemelor ce urmează:
 - a) memorarea fișierelor pe benzi magnetice (exercițiul 5, paragraful 5.3);
 - b) problema continuă a rucsacului (exercițiul 6, paragraful 5.3);
 - c) colorarea unei hărți (exercițiul 9, paragraful 5.4);
 - d) problema discretă a rucsacului (exercițiul 7, paragraful 5.6);
 - e) arhivarea fișierelor (exercițiul 9, paragraful 5.6);
 - f) triangularea polinoamelor (exercițiul 10, paragraful 5.6);
 - g) jocul Perspico (paragraful 5.8);
 - h) problema comis-voiajorului (exercițiul 10, paragraful 5.8).

ALGORITMI DE REZOLVARE A UNOR PROBLEME MATEMATICE

6.1. Operații cu mulțimi

Generarea soluțiilor posibile ale unei probleme presupune prelucrarea elementelor ce aparțin diferitor mulțimi.

Fie A o mulțime oarecare cu n elemente:

$$A = \{a_1, a_2, \dots, a_j, \dots, a_n\}.$$

Întrucât limbajul PASCAL nu permite accesul direct la elementele mulțimilor descrise cu ajutorul tipului de date `set`, vom memora elementele mulțimii A într-un vector (tablou unidimensional) cu n componente $\mathbf{A} = (a_1, a_2, \dots, a_j, \dots, a_n)$. Evident, o astfel de reprezentare stabilește implicit o ordine a elementelor mulțimii corespunzătoare ordinii în care apar componentele în vectorul \mathbf{A} .

Fie A_i o submulțime a lui A . În PASCAL această submulțime poate fi reprezentată prin **vectorul caracteristic al submulțimii**:

$$B_i = (b_1, b_2, \dots, b_j, \dots, b_n),$$

unde

$$b_j = \begin{cases} 1, & \text{dacă } a_j \in A_i; \\ 0, & \text{în caz contrar.} \end{cases}$$

Între submulțimile A_i ale mulțimii A și vectorii caracteristici B_i există o corespondență biunivocă:

$$\begin{array}{ll} A_1 = \emptyset & \leftrightarrow B_1 = (0, 0, \dots, 0); \\ A_2 = \{a_1\} & \leftrightarrow B_2 = (1, 0, \dots, 0); \\ A_3 = \{a_2\} & \leftrightarrow B_3 = (0, 1, \dots, 0); \\ A_4 = \{a_1, a_2\} & \leftrightarrow B_4 = (1, 1, \dots, 0); \\ \dots & \dots \\ A_k = \{a_1, a_2, \dots, a_n\} & \leftrightarrow B_k = (1, 1, \dots, 1); \end{array}$$

Evident, numărul tuturor submulțimilor unei mulțimi este $k = 2^n$.

Reprezentarea submulțimilor cu ajutorul vectorilor caracteristici presupune elaborarea unei unități de program ce conține câte o procedură pentru fiecare din operațiile frecvent utilizate în calculul cu mulțimi: \cup - reuniunea, \cap - intersecția, \setminus - diferența, $\bar{}$ - complementarea. Scrierea procedurilor respective este propusă cititorului în calitate de exercițiu.

O altă operație frecvent utilizată în algoritmi bazați pe metoda trierii este **generarea tuturor submulțimilor** unei mulțimi. Realizarea acestei operații în programele PASCAL este prezentată cu ajutorul exemplului ce urmează.

Exemplul 1. Se consideră mulțimea $A = \{a_1, a_2, \dots, a_n\}$ formată din n numere întregi. Determinați dacă există cel puțin o submulțime A_i , $A_i \subseteq A$, suma elementelor căreia este egală cu m .

Rezolvare. Soluțiile posibile $\emptyset, \{a_1\}, \{a_2\}, \{a_1, a_2\}$ ș.a.m.d. pot fi generate formând consecutiv vectorii binari B_1, B_2, \dots, B_k .

```

Program P161;
  { Generarea tuturor submulțimilor unei mulțimi }
const nmax=50;
type Multime = array [1..nmax] of integer;
      CifraBinara = 0..1;
      VectorCaracteristic = array[1..nmax] of CifraBinara;
var A : Multime;
      B : VectorCaracteristic;
      n, m, j : integer;

function SolutiePosibila : boolean;
var j, suma : integer;
begin
  suma:=0;
  for j:=1 to n do
    if B[j]=1 then suma:=suma+A[j];
  if suma=m then SolutiePosibila:=true
    else SolutiePosibila:=false;
end; { SolutiePosibila }

procedure PrelucrareaSolutiei;
var j : integer;
begin
  write('Submulțimea: ');
  for j:=1 to n do
    if B[j]=1 then write(A[j], ' ');
  writeln;
end; { PrelucrareaSolutiei }

procedure GenerareSubmultimi(var t:CifraBinara);
var j : integer;
begin
  t:=1; { transportul }
  for j:=1 to n do
    if t=1 then
      if B[j]=1 then B[j]:=0
        else begin B[j]:=1; t:=0 end;
end; { GenerareSubmultimi }

```

```

procedure CautareSubmultimi;
var i : integer;
    t : CifraBinara;
begin
    for j:=1 to n do B[j]:=0;
    { începem cu vectorul caracteristic B=(0, 0, ..., 0) }
    repeat
        if SolutiePosibila then PrelucrareaSolutiei;
        GenerareSubmultimi(t);
    until t=1;
end; { CautareSubmultimi }

begin
    write('Dați n='); readln(n);
    writeln('Dați ', n, ' numere întregi:');
    for j:=1 to n do read(A[j]);
    write('Dați m='); readln(m);
    CautareSubmultimi;
    writeln('Sfârșit');
    readln;
end.

```

În programul P161 trierea consecutivă a soluțiilor posibile este organizată cu ajutorul ciclului **repeat...until** din componența procedurii CautareSubmultimi. Vectorii caracteristici ai submulțimilor respective sînt formați cu ajutorul procedurii GenerareSubmultimi. În această procedură vectorul caracteristic B este tratat ca un număr binar valoarea căruia trebuie mărită cu o unitate la fiecare apel.

Instrucțiunile **if** din componența procedurii GenerareSubmultimi simulează funcționarea unui semisumator care adună cifrele binare b_j și t , variabila t reprezentînd transportul. Valoarea $t = 1$ a transportului din rangul n indică faptul că de la vectorul final $B_k = (1, 1, \dots, 1)$ se trece la vectorul inițial $B_1 = (0, 0, \dots, 0)$.

Complexitatea temporală a algoritmilor bazați pe generarea tuturor submulțimilor unei mulțimi este $O(2^n)$.

În unele probleme mulțimea soluțiilor posibile S poate fi calculată ca **produsul cartezian al altor mulțimi**.

Exemplul 2. Se consideră n mulțimi A_1, A_2, \dots, A_n , fiecare mulțime A_j fiind formată din m_j numere întregi. Selectați din fiecare mulțime A_j cîte un număr a_j în așa mod, încît produsul $a_1 \times a_2 \times \dots \times a_n$ să fie maxim.

Rezolvare. Mulțimea soluțiilor posibile $S = A_1 \times A_2 \times \dots \times A_n$. Evident, numărul soluțiilor posibile $k = m_1 \times m_2 \times \dots \times m_n$. Fiecare element s_i al produsului cartezian $A_1 \times A_2 \times \dots \times A_n$ poate fi reprezentat **prin vectorul indicilor**:

$$C_i = (c_{1i}, c_{2i}, \dots, c_{ji}, \dots, c_{ni}),$$

unde c_j este indicele elementului respectiv din mulțimea A_j . De exemplu, pentru

$$\begin{array}{ccc} \textcircled{1} & \textcircled{2} & \textcircled{3} \\ A_1 = (-6, 2, 1); & A_2 = (4, 9); & A_3 = (-8, 3, 5), \end{array}$$

obținem:

$$\begin{aligned}s_1 &= (-6, 4, -8) \leftrightarrow C_1 = (\textcircled{1}, \textcircled{1}, \textcircled{1}); \\s_2 &= (2, 4, -8) \leftrightarrow C_2 = (\textcircled{2}, \textcircled{1}, \textcircled{1}); \\s_3 &= (1, 4, -8) \leftrightarrow C_3 = (\textcircled{3}, \textcircled{1}, \textcircled{1}); \\s_4 &= (-6, 9, -8) \leftrightarrow C_4 = (\textcircled{1}, \textcircled{2}, \textcircled{1}); \\s_5 &= (2, 9, -8) \leftrightarrow C_5 = (\textcircled{2}, \textcircled{2}, \textcircled{1}); \\&\dots \\s_{18} &= (1, 9, 5) \leftrightarrow C_{18} = (\textcircled{3}, \textcircled{2}, \textcircled{3}),\end{aligned}$$

unde $\textcircled{1}$, $\textcircled{2}$ și $\textcircled{3}$ sînt indicii elementelor din mulțimile respective.

Vectorii C_1, C_2, \dots, C_k pot fi generați în ordine lexicografică pornind de la vectorul inițial $C_1=(1, 1, \dots, 1)$.

```
Program P162;
{ Generarea elementelor unui produs cartezian }
const nmax=50; { numărul maximal de mulțimi }
      mmax=50; { numărul maximal de elemente }
type Multime = array [1..mmax] of integer;
      VectorIndicii = array[1..nmax] of 1..mmax;

var A : array[1..nmax] of Multime;
    n : 1..nmax; { numărul de mulțimi }
    M : array[1..nmax] of 1..mmax; { cardinalul mulțimii A[i] }
    Pmax : integer; { produsul maximal }
    C, Cmax : VectorIndicii;
    i, j : integer;

procedure PrelucrareaSolutieiPosibile;
var j, p : integer;
begin
    p:=1;
    for j:=1 to n do p:=p*A[j, C[j]];
    if p > Pmax then begin Pmax:=p; Cmax:=C end;
end; { PrelucrareaSolutieiPosibile }

procedure GenerareProdusCartezian(var t:integer);
var j : integer;
begin
    t:=1; { transportul }
    for j:=1 to n do
        begin
            C[j]:=C[j]+t;
            if C[j]<=M[j] then t:=0 else C[j]:=1;
        end; { for }
    end; { GenerareProdusCartezian }
```

```

procedure CautareaProdusuluiMaximal;
var j : integer;
    t : integer;
begin
    Pmax:=-MaxInt;
    writeln('Pmax=', Pmax);
    for j:=1 to n do C[j]:=1;
    { începem cu vectorul indiciilor C=(1, 1, ..., 1) }
    repeat
        PrelucrareaSolutieiPosibile;
        write('Produsul cartezian: ');
        for j:=1 to n do write(A[j, C[j]], ' '); writeln;
        GenerareProdusCartezian(t);
    until t=1;
end; { CautareaProdusuluiMaximal }

begin
    write('Dați numărul de mulțimi n='); readln(n);
    for i:=1 to n do
        begin
            write('Dați cardinalul M[' , i, ']='); readln(M[i]);
            write('Dați elementele multimii A[' , i, ']: ');
            for j:=1 to M[i] do read(A[i, j]);
            writeln;
        end;

    CautareaProdusuluiMaximal;

    writeln('Pmax=', Pmax);
    write('Elementele selectate: ');
    for j:=1 to n do write(A[j, Cmax[j]], ' ');
    writeln;
    readln;
    readln;
end.

```

În procedura GenerareProdusCartezian vectorul indiciilor C este tratat ca un număr natural scris într-un **sistem mixt de numerație**. În acest sistem cifra c_1 este scrisă în baza m_1 , cifra c_2 - în baza m_2 , cifra c_3 - în baza m_3 ș.a.m.d. La fiecare apel al procedurii GenerareProdusCartezian valoarea numărului înscris în vectorul C este mărită cu o unitate. Valoarea $t = 1$ a transportului din rangul n indică faptul că de la vectorul final $C_k = (m_1, m_2, \dots, m_n)$ se trece la vectorul inițial $C_1 = (1, 1, \dots, 1)$.

Menționăm că dacă numărul de mulțimi n este cunoscut pînă la scrierea programului, generarea elementelor produsului cartezian $A_1 \times A_2 \times \dots \times A_n$ poate fi făcută mult mai simplu cu ajutorul a n cicluri imbricate:

```

for j1:=1 to m1 do
  for j2:=1 to m2 do
    ...
    for jn:=1 to mn do
      if SolutiePosibila(aj1, aj2, ..., ajn)
        then PrelucrareaSoluției(aj1, aj2, ..., ajn)

```

Complexitatea temporală a algoritmilor bazați pe generarea tuturor elementelor unui produs cartezian este $O(m^n)$, unde $m = \max(m_1, m_2, \dots, m_n)$.

Întrebări și exerciții

- ❶ Submulțimile A_i, A_j ale mulțimii A sînt reprezentate prin vectori caracteristici. Elaborați procedurile necesare pentru efectuarea următoarelor operații: $A_i \cap A_j, A_i \cup A_j, A_i \setminus A_j, \bar{A}_i$.
- ❷ Orice submulțime $A_i, A_j \subseteq A$, poate fi reprezentată printr-un vector cu n componente, unde elementele submulțimii A_i sînt plasate la începutul vectorului, iar restul pozițiilor au o valoare ce nu aparține mulțimii A . Elaborați procedurile necesare pentru efectuarea operațiilor frecvent întîlnite în calculul cu mulțimi: $\cup, \cap, \setminus, \bar{}$. Cum credeți, care reprezentare a submulțimilor este mai comodă: prin vectorii caracteristici sau prin vectorii ce conțin chiar elementele submulțimii?
- ❸ Estimați timpul de execuție a procedurii `CautareSubmultimi` din programul P161. Verificați aceste estimări prin măsurări directe ale timpului de execuție pentru diferite valori ale lui n .
- ❹ Se consideră mulțimea A formată din primele n caractere ale alfabetului latin. Elaborați un program care afișează la ecran toate submulțimile acestei mulțimi.
- ❺ Se consideră numărul natural $n = 32*35*17^*$, format din 9 cifre zecimale. Determinați cifrele care trebuie înscrise în pozițiile marcate cu simbolul $*$ pentru ca numărul obținut să se împartă fără rest la m .
- ❻ Estimați timpul de execuție a procedurii `CautareaProdusuluiMaximal` din programul P162. Verificați aceste estimări prin măsurări directe ale timpului de execuție pentru diferite valori ale lui n și m_1, m_2, \dots, m_n .
- ❼ Într-un coș sînt m mere și p pere. Să se genereze toate posibilitățile de a alege f fructe dintre care k să fie mere.
- ❽ Elaborați o procedură recursivă care generează toate elementele unui produs cartezian.
- ❾ Fie $A = (a_1, a_2, \dots, a_p, \dots, a_n)$ o mulțime ordonată de caractere numită **alfabet**. Numim **cuvînt de lungime p** orice succesiune de p caractere din alfabetul A . Elaborați o procedură care generează toate cuvintele de lungimea p .

6.2. Analiza combinatorie

În rezolvarea multor probleme implementarea algoritmilor bazați pe analiza consecutivă a soluțiilor posibile presupune generarea permutărilor, aranjamentelor sau combinărilor unei mulțimi.

Generarea permutărilor. E cunoscut faptul că numărul de permutări posibile ale unei mulțimi $A=\{a_1, a_2, \dots, a_n\}$ cu n elemente se determină ca $P_n = n!$. Acest număr poate fi calculat cu ajutorul funcției *factorial*, exprimată în formă iterativă:

$$P_n = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

sau recursivă:

$$P_n = \begin{cases} 1, & \text{dacă } n = 0; \\ P_{n-1} \cdot n, & \text{dacă } n > 0. \end{cases}$$

De exemplu, pentru $A=\{a_1, a_2\}$ cele $P_2 = 2! = 2$ permutări sînt (a_1, a_2) și (a_2, a_1) . Pentru $A=\{a_1, a_2, a_3\}$ cele $P_3 = 3! = 6$ permutări sînt:

$$\begin{array}{lll} (a_1, a_2, a_3); & (a_1, a_3, a_2); & (a_2, a_1, a_3); \\ (a_2, a_3, a_1); & (a_3, a_1, a_2); & (a_3, a_2, a_1). \end{array}$$

Întrucît între permutările mulțimii $A=\{a_1, a_2, \dots, a_n\}$ și permutările mulțimii $I=\{1, 2, \dots, n\}$ există o corespondență biunivocă, problema generării permutărilor oricărei mulțimi A cu n elemente se reduce la generarea permutărilor mulțimii $\{1, 2, \dots, n\}$, denumite **permutări de grad n** .

Există mai multe metode ingenioase de generare a permutărilor de grad n , cea mai răspîndită fiind **metoda lexicografică**. În această metodă se pleacă de la permutarea cea mai mică în ordine lexicografică, și anume de la **permutarea identică** $(1, 2, \dots, n)$.

Avînd construită o permutare $p = (p_1, \dots, p_{i-1}, p_i, p_{i+1}, \dots, p_n)$, pentru determinarea următoarei permutări p' care îi urmează în ordine lexicografică se caută acel indice i care satisface relațiile:

$$p_i < p_{i+1} < p_{i+2} < \dots < p_n.$$

În continuare, elementul p_i este înlocuit cu cel mai mic dintre elementele p_{i+1}, \dots, p_n care este mai mare decît p_i , fie el p_k :

$$(p_1, \dots, p_{i-1}, p_k, p_{i+1}, \dots, p_{k-1}, p_i, p_{k+1}, \dots, p_n).$$

Permutarea căutată p' se obține prin inversarea ordinii ultimilor $(n - i)$ elemente din acest vector, astfel încît ele să apară în ordine crescătoare.

Dacă nu există niciun indice i ca mai sus, înseamnă că s-a ajuns la permutarea cea mai mare în ordine lexicografică, adică la $(n, (n - 1), \dots, 1)$ și algoritmul se termină.

De exemplu, pentru $n=3$ se obțin permutările:

$$\begin{array}{lll} (1, 2, 3); & (1, 3, 2); & (2, 1, 3); \\ (2, 3, 1); & (3, 1, 2); & (3, 2, 1). \end{array}$$

În programul ce urmează metoda lexicografică este realizată cu ajutorul procedurii `GenerarePermutari`.

```
Program P163;
{ Generarea permutărilor }
const nmax=100;

type Permutare=array[1..nmax] of 1..nmax;
```



```

var P : Permutare;
    n : 2..nmax;
    Indicator : boolean;
    i : integer;

procedure GenerarePermutari(var Indicator : boolean);
label 1;
var i, j, k, aux : integer;
begin
    { permutarea identică }
    if not Indicator then
        begin
            for i:=1 to n do P[i]:=i;
            Indicator:=true;
            goto 1;
        end;

    { căutarea indicelui i }
    i:=n-1;
    while P[i]>P[i+1] do
        begin
            i:=i-1;
            if i=0 then
                begin
                    { un astfel de indice nu mai există }
                    Indicator:=false;
                    goto 1;
                end; { then }
            end; {while }

    { căutarea indicelui k }
    k:=n;
    while P[i]>P[k] do k:=k-1;

    { interschimbarea P[i] - P[k] }
    aux:=P[i]; P[i]:=P[k]; P[k]:=aux;

    { ordonarea ultimilor (n-i) elemente }
    for j:=1 to (n-i) div 2 do
        begin
            aux:=P[i+j];
            P[i+j]:=P[n-j+1];
            P[n-j+1]:=aux;
        end; { for }
    Indicator:=true;
1:end; { GenerarePermutari }

```

```

begin
write('Dați n='); readln(n);
Indicator:=false;
repeat
  GenerarePermutari(Indicator);
  if Indicator then
    for i:=1 to n do write(P[i] : 3);
    writeln;
  until not Indicator;
readln;
end.

```

Pentru a porni de la permutarea inițială, înainte de primul apel al procedurii *GenerarePermutari*, parametrului *Indicator* i se atribuie valoarea *false*. La fiecare apel procedura înscrie în vectorul *P* permutarea ce urmează în ordine lexicografică și atribuie parametrului *Indicator* valoarea *true*. După generarea tuturor permutărilor, procedura *GenerarePermutari* va atribui parametrului *Indicator* valoarea *false*.

Din păcate, indiferent de metoda folosită, timpul necesar pentru generarea tuturor permutărilor este cel puțin $O(n!)$. În consecință, algoritmi bazați pe căutarea soluțiilor prin generarea tuturor permutărilor posibile pot fi aplicați numai pentru valori mici ale lui *n*.

Generarea aranjamentelor. Numărul aranjamentelor de *m* elemente ale unei mulțimi $A=\{a_1, a_2, \dots, a_n\}$ cu *n* elemente este dat de formula:

$$A_n^m = \frac{n!}{(n-m)!}$$

sau, pentru a evita folosirea funcției *factorial*,

$$A_n^m = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-m+2) \cdot (n-m+1).$$

Ca și în cazul permutărilor, putem reduce problema generării aranjamentelor unei mulțimi arbitrare *A* la generarea aranjamentelor mulțimii $I=\{1, 2, \dots, n\}$. De exemplu, pentru $I=\{1, 2, 3\}$ și $m=2$ cele $A_3^2 = 6$ aranjamente sînt:

$$(1, 2); \quad (2, 1); \quad (1, 3); \\ (3, 1); \quad (2, 3); \quad (3, 2).$$

Generarea aranjamentelor poate fi făcută în ordine lexicografică, pornind de la aranjamentul cel mai mic $a=(1, 2, \dots, m)$.

Fie $a=(a_1, a_2, \dots, a_i, \dots, a_m)$ un aranjament oarecare. Pentru a determina succesorul a' al aranjamentului *a*, se caută mai întii cel mai mare indice *i* cu proprietatea că a_i poate fi mărit. Un element a_i poate fi mărit dacă cel puțin una din valorile a_i+1, a_i+2, \dots, n cu care ar putea fi înlocuit a_i este disponibilă. Pentru a putea efectua mai ușor aceste verificări, se utilizează vectorul $D=(d_1, d_2, \dots, d_i, \dots, d_n)$, unde d_i este 0 sau 1 în funcție dacă valoarea *i* apare sau nu în aranjamentul curent *a*. În momentul în care a fost determinat indicele *i*, elementele a_i, a_{i+1}, \dots, a_m vor primi în ordine crescătoare cele mai mici numere disponibile.

Dacă nu există un indice i cu proprietatea menționată, înseamnă că s-a ajuns la aranjamentul $(n-m+1, n-m+2, \dots, n)$, deci procesul generării s-a încheiat. Pentru a semnaliza acest lucru, în programul ce urmează se utilizează variabila booleană Indicator.

```

Program P164;
  { Generarea aranjamentelor }
const nmax=100;
        mmax=100;

type Aranjament=array[1..mmax] of 1..nmax;

var A : Aranjament;
    D : array[1..nmax] of 0..1;
    n : 1..nmax;
    m : 1..nmax;
    i : integer;
    Indicator : boolean;

procedure GenerareAranjamente(var Indicator : boolean);
label l;
var i, j, k, l : integer;
begin
  { aranjamentul inițial }
  if not Indicator then
    begin
      for i:=1 to m do
        begin
          A[i]:=i; D[i]:=1;
        end;
      for i:=m+1 to n do D[i]:=0;
      Indicator:=true;
      goto l;
    end;
  { succesorul aranjamentului curent }
  for i:=m downto 1 do
    begin
      D[A[i]]:=0;
      for j:=A[i]+1 to n do
        if D[j]=0 then
          begin
            A[i]:=j; D[j]:=1; k:=0;
            for l:=i+1 to m do
              begin
                repeat k:=k+1 until D[k]=0;
                A[l]:=k; D[k]:=1;
              end; { for }
          end;
    end;

```

```

        goto l;
    end; { if }
end; { for }
Indicator:=false;
1:end; { GenerareAranjamente }

begin
write('Dați n='); readln(n);
write('Dați m='); readln(m);
Indicator:=false;
repeat
    GenerareAranjamente(Indicator);
    if Indicator then
        for i:=1 to m do write(A[i] : 3);
        writeln;
    until not Indicator;
    readln;
end.

```

Așadar, complexitatea temporală a algoritmilor bazați pe generarea tuturor aranjamentelor este cel mult $O(n!)$.

Generarea combinațiilor. Numărul de combinații de n elemente luate câte m ($m \leq n$) se calculează ca

$$C_n^m = \frac{A_n^m}{P_m} = \frac{n!}{m!(n-m)!}.$$

De exemplu, pentru $I=\{1, 2, 3\}$ și $m=2$ avem $C_3^2 = 3$ combinații:

$$\{1, 2\}; \quad \{1, 3\}; \quad \{2, 3\}.$$

Generarea combinațiilor poate fi făcută în ordine lexicografică, pornind de la combinația inițială $\{1, 2, \dots, m\}$.

Fie dată o combinație $c=\{c_1, c_2, \dots, c_i, \dots, c_m\}$. Combinația c' care îi urmează imediat în ordine lexicografică se determină astfel:

– se stabilește indicele i care satisface relațiile $c_i < n-m+1$, $c_{i+1} = n-m+i+1$, $c_{m-1} = n-1$, $c_m = n$;

– se trece la combinația $c'=\{c_1, \dots, c_{i-1}, c_i+1, c_i+2, \dots, c_i+n-i+1\}$.

Dacă nu există un indice i care satisface condițiile de mai sus, înseamnă că au fost generate toate combinațiile.

În programul ce urmează combinațiile mulțimii $I=\{1, 2, \dots, n\}$ se generează consecutiv în vectorul (tabloul unidimensional) C .

```

Program P165;
    { Generarea combinarilor }
const nmax=100;
        mmax=100;

type Combinare=array[1..mmax] of 1..nmax;

```

```

var C : Combinare;
      n : 1..nmax;
      m : 1..mmax;
      i : integer;
      Indicator : boolean;

procedure GenerareCombinari(var Indicator : boolean);
label 1;
var i, j : integer;
begin
  { combinarea initiala }
  if not Indicator then
    begin
      for i:=1 to m do C[i]:=i;
      Indicator:=true;
      goto 1;
    end;
  { succesorul combinarii curente }
  for i:=m downto 1 do
    if C[i]<(n-m+i) then
      begin
        C[i]:=C[i]+1;
        for j:=i+1 to m do C[j]:=C[j-1]+1;
        goto 1;
      end; { then }
  Indicator:=false;
1:end; { GenerareCombinari }

begin
  write('Dați n='); readln(n);
  write('Dați m='); readln(m);
  Indicator:=false;
  repeat
    GenerareCombinari(Indicator);
    if Indicator then
      for i:=1 to m do write(C[i] :3);
    writeln;
  until not Indicator;
  readln;
end.

```

Se poate demonstra că timpul cerut de un algoritm bazat pe generarea tuturor combinațiilor este de ordinul $O(n^k)$, unde $k=\min(m, n-m+1)$, deci polinomial.

Întrebări și exerciții

- ❶ Scrieți un program PASCAL care afișează la ecran numărul permutărilor P_n , numărul aranjamentelor A_n^m și numărul combinațiilor C_n^m . Valorile n și m se citesc de la tastatură.
- ❷ Elaborați o procedură recursivă pentru generarea tuturor permutărilor posibile ale mulțimii $I=\{1, 2, \dots, n\}$.
- ❸ Utilizînd metoda reluării, schițați trei algoritmi pentru generarea, respectiv, a permutărilor, aranjamentelor și combinațiilor unei mulțimi formate din n elemente distincte.
- ❹ Se consideră un tablou bidimensional $T[1..n, 1..n]$ format din numere întregi. Elaborați un program care determină o permutare a coloanelor tabloului astfel încît suma componentelor de pe diagonala principală să fie minimă.
- ❺ Elaborați un program care afișează la ecran toate șirurile posibile formate din caracterele A, b, C, d, E. Fiecare caracter apare în șir numai o singură dată.
- ❻ Dintr-o listă ce conține n candidați trebuie alese m persoane care vor fi incluse în echipa de fotbal a unui raion. Elaborați un program care afișează la ecran toate modalitățile de selecție a celor m persoane.
- ❼ Se consideră mulțimea numerelor întregi $A=\{a_1, a_2, \dots, a_n\}$. Elaborați un program care determină o submulțime ce conține exact m elemente ale mulțimii A astfel încît suma lor să fie maximă.
- ❽ Cum credeți, care sînt avantajele și neajunsurile algoritmilor bazați pe generarea tuturor permutărilor, aranjamentelor și combinațiilor posibile?
- ❾ Există oare tehnici de programare care ar permite evitarea unei analize exhaustive a tuturor permutărilor, aranjamentelor sau combinațiilor posibile?
- ❿ Se consideră mulțimea numerelor întregi $A=\{a_1, a_2, \dots, a_n\}$. Elaborați un program care determină o descompunere a mulțimii A în două submulțimi nevide B și C astfel încît suma elementelor din submulțimea B să fie egală cu suma elementelor din submulțimea C . De exemplu, pentru $A=\{-4, -1, 0, 1, 2, 3, 9\}$ avem $B=\{-4, 0, 9\}$ și $C=\{-1, 1, 2, 3\}$.

PROBLEME RECAPITULATIVE

Problemele ce urmează au fost propuse la diverse concursuri de informatică. Rezolvarea lor necesită cunoașterea profundă a tehnicilor de programare și a metodelor de estimare a complexității algoritmilor.

1. Cercuri. Un pătrat cu latura a cm conține n cercuri ($n \leq 100$). Fiecare cerc i este definit prin coordonatele centrului (x_i, y_i) și raza r_i . Elaborați un program care în cel mult t secunde calculează cât mai exact aria obținută prin reuniunea celor n cercuri.

2. Puncte. Se dau n puncte în plan, $n \leq 100$. Să se calculeze numărul maxim de puncte coliniare.

3. Poduri. Se consideră n insule legate prin m poduri. E cunoscut faptul că pe podul ce leagă insulele i, j pot circula vehicule greutatea cărora nu depășește g_{ij} tone. Determinați greutatea maximă G_{ab} a vehicolului care poate ajunge de pe insula a pe insula b .

4. Texte. Se consideră n texte care trebuie tipărite pe foi de hîrtie. Textul i este format din r_i linii. Pe o foaie pot fi tipărite cel mult m linii care pot forma unul, două sau mai multe texte. Pentru a le separa, între textele de pe aceeași foaie se inserează o linie vidă. Fragmentarea textelor este interzisă, adică toate liniile oricărui text trebuie imprimate pe aceeași foaie. Elaborați un program care determină numărul minim de foi necesare pentru a tipări toate textele.

5. Circumferințe. Se consideră n puncte pe un plan cartezian. Fiecare punct i este definit prin coordonatele sale x_i, y_i . Elaborați un program care verifică dacă punctele în studiu aparțin unei circumferințe.

6. Rețeaua telefonică. Se consideră n orașe, $n \leq 100$, care trebuie legate prin cabluri într-o rețea telefonică. Pentru fiecare oraș i sînt cunoscute coordonatele carteziene x_i, y_i . Cablurile telefonice ce leagă oricare două orașe nu pot avea ramificații. Abonații rețelei telefonice comunică între ei direct sau prin intermediul stațiilor telefonice din alte orașe. Lungimea cablului care leagă orașele i, j este egală cu distanța dintre ele. Determinați lungimea sumară minimă a cablurilor necesare pentru a conecta toate orașele.

7. Evaluarea expresiilor. Se consideră expresiile aritmetice formate din numere întregi, parantezele $(,)$ și operatorii binari $+, -, *, \text{mod}, \text{div}$. Scrieți un program care evaluează expresiile aritmetice în studiu.

8. Psihologie. Se consideră n angajați, $n \leq 100$, care trebuie repartizați în m echipe. Fiecare echipă este formată din k angajați, $k \cdot m = n$. Relația de compatibilitate între angajații i, j se caracterizează prin coeficientul r_{ij} care poate lua valorile 0 (incompatibili), 1, 2, ..., 10 (compatibilitate excelentă). Compatibilitatea întregului colectiv C

se calculează însumînd coeficienții r_{ij} pentru toate perechile posibile (i, j) din cadrul fiecărei echipe. Determinați compatibilitatea maximă C_{max} care poate fi asigurată prin formarea corespunzătoare a echipelor.

9. Compasul. Se consideră n puncte pe un plan cartezian. Fiecare punct i este specificat prin coordonatele sale x_i, y_i . Elaborați un program care verifică dacă se poate desena o circumferință cu centrul în unul din punctele în studiu și care ar trece prin toate celelalte puncte.

10. Intersecția dreptunghiurilor. Se dau n dreptunghiuri ($n \leq 10$) care au laturile paralele cu axele de coordonate, iar coordonatele vîrfurilor sînt numere naturale din mulțimea $\{0, 1, 2, \dots, 20\}$. Elaborați un program care calculează aria figurii obținute prin intersecția celor n dreptunghiuri.

11. Reuniunea dreptunghiurilor. Se dau n dreptunghiuri ($n \leq 10$) care au laturile paralele cu axele de coordonate, iar coordonatele vîrfurilor sînt numere reale. Elaborați un program care calculează aria figurii obținute prin reuniunea celor n dreptunghiuri.

12. Numere prime. Calculați toate numerele prime formate din 4 cifre inversul cărora este la fel un număr prim, iar suma cifrelor este de asemenea un număr prim.

13. Vizibilitate. Se consideră linia frîntă închisă $P_1P_2 \dots P_nP_1$, $n \leq 20$, care nu se autointersectează. În punctul A din interiorul liniei este situat un observator. Pentru observator unele segmente ale liniei frînte pot fi invizibile. Elaborați un program care calculează numărul segmentelor invizibile.

14. Felinare. Un parc de formă dreptunghiulară este împărțit în pătrate de aceeași dimensiuni. În fiecare pătrat al parcului poate fi instalat cîte un felinar. În general, un felinar asigură iluminarea nu numai a pătratului în care el se află, dar și a celor opt pătrate vecine. Elaborați un program care determină numărul minim de felinare necesare pentru iluminarea parcului.

15. Laserul. Se consideră o placă dreptunghiulară cu dimensiunile $m \times n$, unde m și n sînt numere naturale. Această placă trebuie tăiată în $m \times n$ plăci mai mici, fiecare bucată avînd forma unui pătrat cu dimensiunile 1×1 . Întrucît placa este neomogenă, pentru fiecare bucată se indică densitatea d_{xy} , unde x, y sînt coordonatele colțului stînga-jos al pătratului respectiv.

Pentru operațiile de tăiere se folosește un strung cu laser. Fiecare operație de tăiere include:

- fixarea unei plăci pe masa de tăiere;
- stabilirea puterii laserului în funcție de densitatea materialului de tăiat;
- o singură deplasare a laserului de-a lungul oricărei drepte paralele cu una din axele de coordonate;
- scoaterea celor două plăci de pe masa de tăiere.

Costul unei operații de tăiere se determină după formula $c = d_{max}$, unde d_{max} este densitatea maximă a bucăților 1×1 peste marginile cărora trece raza laserului. Evident, costul total T poate fi determinat adunînd costurile individuale c ale tuturor operațiilor de tăiere necesare pentru obținerea bucăților 1×1 . Scrieți un program care calculează costul minim T .

16. Județe. Teritoriul unei țări este împărțit în județe (fig. 7.1). Pe hartă, frontiera țării și granițele administrative ale fiecărui județ reprezintă câte un poligon definit prin coordonatele (x_i, y_i) ale vîrfurilor sale. Se presupune că vîrfurile de poligoane sînt numerotate direct prin 1, 2, 3, ..., n , iar coordonatele lor sînt numere întregi. În interiorul oricărui județ nu există alte județe.

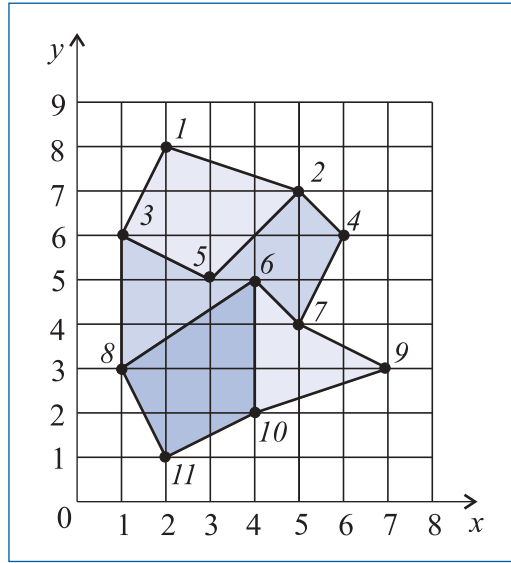


Fig. 7.1. Județele unei țări

Un virus de calculator a distrus parțial informația despre granițele administrative, lăsînd intacte următoarele date:

- numărul n și coordonatele (x_i, y_i) ale tuturor vîrfurilor de poligoane;
- numărul de segmente m care formează laturi de poligoane și informația despre extremitățile fiecărui segment.

Scrieți un program care determină numărul de județe d și vîrfurile fiecărui poligon ce reprezintă o graniță administrativă de județ.

17. Turnuri. Fie n plăci dreptunghiulare numerotate de la 1 la n . Despre placa i se știe că are grosimea h_i și lungimile laturilor x_i, y_i . Elaborați un program care determină înălțimea maximă a unui turn ce poate fi construit din plăcile în studiu. Pentru a asigura stabilitatea turnului, se vor respecta următoarele reguli (fig. 7.2):

- plăcile sînt puse una peste alta orizontal, nu pe muchii sau în alt mod;
- muchiile omoloage ale plăcilor suprapuse sînt paralele;
- orice placă din componența turnului se va sprijini în întregime pe placa de mai jos (evident, placa de la baza tunului se sprijină pe sol);
- nu se cere să folosim toate plăcile.

Date de intrare. Fișierul text TURNURI . IN conține pe prima linie numărul n . Pe fiecare din următoarele n linii se conțin câte trei numere întregi pozitive x_i, y_i, h_i separate prin spațiu.

Date de ieșire. Fișierul text TURNURI . OUT va conține pe o singură linie un număr întreg – înălțimea maximă a turnului.

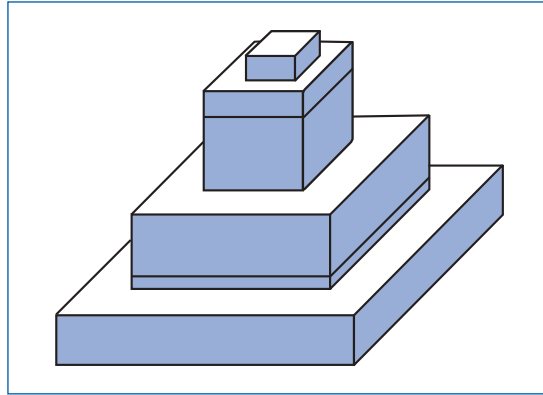


Fig. 7.2. Turn construit din plăci dreptunghiulare

Exemplu:

TURNURI . IN	TURNURI . OUT
5 3 4 2 3 4 3 4 3 2 1 5 4 2 2 1	8

Restricții: $n \leq 1000$; $x_i, y_i, h_i < 1000$. Timpul de execuție nu va depăși 10 sec.

18. Speologie. Speologul este un specialist care se ocupă cu explorarea și studierea peșterilor. Antrenarea speologilor presupune parcurgerea unor labirinturi subterane. Un astfel de labirint constă din n , $n \leq 100$, peșteri și coridoare (fig. 7.3). Fiecare peșteră are o denumire individuală formată din cel mult 10 caractere alfanumerice, fără spații, scrisă pe unul din pereții ei. Peștera de intrare are denumirea *INTRARE*, iar cea de ieșire – denumirea *IESIRE*. La intrarea în fiecare coridor este scrisă denumirea peșterii spre care ea duce.

Speologul nu cunoaște planul labirintului. În schimb, el este echipat cu un caiet, un creion și o lanternă, fapt ce îi permite să poată citi denumirile de peșteri sau de coridoare și să facă notițe. Vom numi drum o succesiune de peșteri cu proprietatea că între oricare două peșteri consecutive din succesiune există un coridor. Prin lungimea drumului înțelegem numărul de peșteri ce-l formează. De exemplu, drumul *INTRARE*, *STALAGMITE*, *LILIECI*, *IZVOARE*, *IESIRE* are lungimea 5.

Elaborați un program care găsește unul dintre cele mai scurte drumuri de la peștera *INTRARE* la peștera *IESIRE*.

Date de intrare. Fișier de intrare nu există. Totuși caracteristica peșterii curente poate fi aflată prin apelul funcției predefinite *UndeMaAflu* de tip *string*. Funcția returnează un șir de caractere ce conține denumirea peșterii în care în prezent se află speologul, două puncte și denumirile de intrări de galerii, separate prin spațiu. De exemplu, dacă speologul se află în peștera *LILIECI*, funcția va întoarce valoarea:

```
LILIECI: STALAGMITE IZVOARE LILIECI LILIECI
```

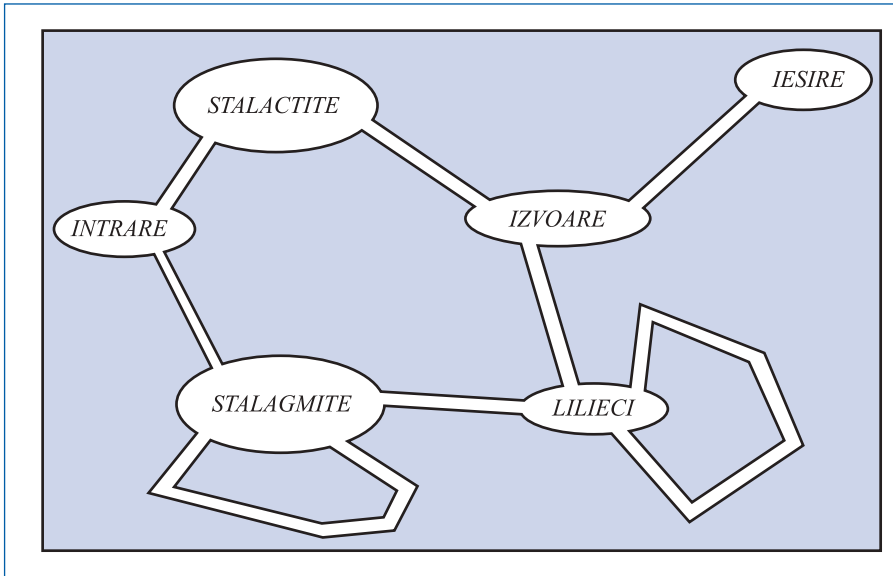


Fig. 7.3. Planul unei peșteri

Trecerea speologului din peștera curentă în peștera spre care duce galeria c se realizează prin apelul procedurii predefinite `TreciCoridorul(c)`, unde c este o expresie de tip `string`. Dacă se indică un coridor inexistent, speologul rămîne pe loc. Pentru ca aceste subprograme să fie accesibile, includeți în partea declarativă a programului de elaborat linia

```
uses LABIRINT;
```

Date de ieșire. Fișierul text `SPEOLOG.OUT` va conține pe prima linie un număr întreg – lungimea celui mai scurt drum. Pe următoarele linii se va scrie drumul. Fiecare denumire de peșteră ocupă o linie separată. Dacă un astfel de drum nu există, fișierul va conține o singură linie cu cuvîntul `FUNDAC`.

Exemplu. Pentru labirintul din figura 7.3 avem:

```
SPEOLOG.OUT
```

```
4
INTRARE
STALACTITE
IZVOARE
IESIRE
```

Timpul de execuție nu va depăși 20 sec.

19. Livada. Planul unei livezi de formă dreptunghiulară cu dimensiunile $n \times m$ este format din zone pătrate cu latura 1. În fiecare zonă crește o singură specie de arbori. Orice specie de arbori poate ocupa una sau mai multe zone, nu neapărat vecine. Elaborați un program care găsește un sector dreptunghiular de arie minimă ce conține cel puțin k specii de arbori. Laturile sectorului vor coincide cu laturile zonelor din plan.

20. Discoteca. La o discotecă participă mai multe persoane, numerotate de la 1 la n . Inițial, numai un singur participant, cel cu numărul i , cunoaște o știre foarte importantă pe care el o comunică prietenilor săi. În continuare, orice participant j , care deja cunoaște această știre, o comunică, de asemenea, numai prietenilor săi. Elaborați un program care determină numărul de participanți p care vor afla știrea respectivă. Relația de prietenie este definită prin m perechi distincte de tipul $\{j, k\}$ cu semnificația „participanții j, k sînt prieteni”. Se consideră că $3 \leq n \leq 1000$ și $2 \leq m \leq 30000$.

21. Genistul. O porțiune de drum este împărțită în n segmente. Pe fiecare segment poate fi plantată cel mult o mină. Genistul a memorat informația despre minele plantate într-un tablou unidimensional $M = \|m_i\|$, în care $m_i = 1$ dacă segmentul i conține o mină și $m_i = 0$ în caz contrar. Pentru orice eventualitate, genistul a cifrat informația din tabloul M într-un alt tablou $C = \|c_i\|_n$ componentele căruia se determină după cum urmează:

$$c_i = \begin{cases} m_1 + m_2, & \text{pentru } i = 1; \\ m_{i-1} + m_i + m_{i+1}, & \text{pentru } 1 < i < n; \\ m_{n-1} + m_n, & \text{pentru } i = n. \end{cases}$$

În condițiile de luptă datele din tabloul M au fost pierdute. Elaborați un program care calculează tabloul M avînd ca date de intrare tabloul C . Se consideră că problema are cel puțin o soluție și $3 \leq n \leq 10000$.

22. Cutii. Se consideră n cutii în formă de paralelipiped dreptunghiular. Pentru fiecare cutie sînt cunoscute cele trei dimensiuni x_i, y_i, z_i . În funcție de dimensiunile cutiilor, unele din ele pot fi puse una în alta. În general, cutiile pot fi rotite. Cutia i poate fi pusă în cutia j numai atunci cînd în urma tuturor rotirilor posibile se va găsi o astfel de poziție pentru care dimensiunile cutiei i sînt cu strictețe mai mici decît dimensiunile corespunzătoare ale cutiei j . Pentru a asigura o împachetare estetică, se cere ca fețele omoloage ale cutiilor să fie paralele (fig. 7.4). Mai mult decît atît, unele cutii pot fi imbricate sau, cu alte cuvinte, poate fi format un șir de numere de cutii $i_1, i_2, i_3, \dots, i_k$ cu proprietatea că cutia i_1 se află în cutia i_2 ; cutia i_2 se află în cutia i_3 etc. Scrieți un program care determină numărul maxim de cutii k_{\max} ce pot fi imbricate într-un astfel de șir.

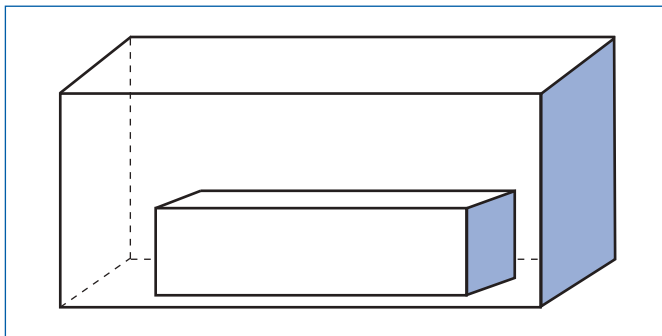


Fig. 7.4. Cutii imbricate

Date de intrare. Fișierul text CUTII . IN conține pe prima linie numărul natural n . Fiecare din următoarele n linii conține câte trei numere naturale x_i, y_i, z_i separate prin spațiu.

Date de ieșire. Fișierul text CUTII . OUT va conține pe o singură linie numărul natural k_{\max} .

Exemplu:

CUTII . IN

```
5
4 4 4
1 3 5
2 2 3
1 1 1
1 1 2
```

CUTII . OUT

```
3
```

Restricții: $2 \leq n \leq 500$; $1 \leq x_i, y_i, z_i \leq 30\,000$. Timpul de execuție nu va depăși 2 secunde.

Bibliografie

1. *Cerchez Emanuela, Șerban Marinel*. Informatică. Manual pentru clasa a X-a. Filiera teoretică, profilul matematică-informatică. Iași: Editura POLIROM, 2000, 200 p.
2. *Cerchez Emanuela*. Informatică. Culegere de probleme pentru liceu. Iași, Editura POLIROM, 2002, 240 p.
3. *Galatan Suzana, Ghinea Diana, Întuneric Ana, Radu Stefana*. Ghid de pregătire pentru BAC. Informatică. Intensiv. Pascal C/C++. București, Editura Sigma, 2009, 539 p.
4. *Giumale Cristian*. Un atelier de programare. Cluj-Napoca. Editura Computer Libris AGORA, 2000, 382 p.
5. *Giumale Cristian*. Introducere în analiza algoritmilor. Iași, Editura POLIROM, 2004, 456 p.
6. *Gremalschi Anatol, Mocanu Iurie, Spinei Ion*. Informatică. Limbajul PASCAL. Manual pentru clasele IX-XI. Chișinău, Editura Știința, 2003, 256 p.
7. *Ivașc Cornelia, Prună Mona*. Bazele informaticii (Grafuri și elemente de combinatorică). Proiect de manual pentru clasa a X-a. Profil informatică. București, Editura Petron, 1995, 175 p.
8. *Livovschi Leon, Georgescu Horia*. Sinteza și analiza algoritmilor. București, Editura Științifică și Pedagogică, 1986, 458 p.
9. *Moraru Florin*. Bacalaureat. Informatică. București, Editura Petron, 2000, 319 p.
10. *Negreanu Dan*. Probleme de matematică rezolvate cu calculatorul. București, Editura Teora, 1998, 214 p.
11. *Roșca Ion Gh., Cocianu Cătălina, Uscatu Cristian*. Bazele informaticii. Manual pentru clasa a X-a, licee teoretice. București, Editura ALL EDUCAȚIONAL, 1999, 64 p.
12. *Roșca Ion Gh., State Luminița, Ghilic-Micu Bogdan, Cocianu Cătălina-Luca, Stoica Marian, Uscatu Cristian*. Informatică. Manual pentru clasa a 10-a. Profilul matematică-informatică. București, ALL EDUCAȚIONAL, 2000, 96 p.
13. *Sorin Tudor*. Informatică (Tehnici de programare). Manual pentru clasa a X-a. Varianta Pascal. București, Editura L&S INFOMAT, 2000, 188 p.
14. *Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest*. Introducere în algoritmi. Cluj-Napoca, Editura Computer Libris AGORA, 2000, 881 p.
15. *Ахо Альфред В., Хопкрофт Джон, Ульман Джеффри Д.* Структуры данных и алгоритмы. Пер. с англ. : Уч. пос. М.: Издательский дом «Вильямс», 2003, 384 стр.
16. *Брайков А.* Turbo PASCAL. Сборник задач. Кишинэу, Издательство Prut International, 2007, 232 стр.
17. *Вирт Н.*, Алгоритмы + Структуры данных = Программы, М., Издательство «Мир», 1985, 243 стр.
18. *Вирт Н.* Алгоритмы и структуры данных, М., Издательство «Мир», 1989, 350 стр.

19. Гремальски А., Мокану Ю., Спиной И. Информатика. Язык программирования ПАСКАЛЬ. Издательство «Штиинца», 2000, 270 стр.
20. Долинский М.С. Решение сложных и олимпиадных задач по программированию: Учебное пособие. СПб.: ПИТЕР, 2006, 366 стр.
21. Йенсен К., Вирт Н. Паскаль, Руководство пользователя. М., Издательство «Финансы и статистика», 1989, 255 стр.
22. Кирюхин В.М. Методика проведения и подготовки к участию в олимпиадах по информатике: Всероссийская олимпиада школьников, М.: БИНОМ. Лаборатория знаний, 2012, 272 стр.
23. Окулов С.М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004, 341 стр.
24. Окулов С.М. Основы программирования. М.: БИНОМ. Лаборатория знаний, 2008, 231 стр.
25. Под ред. Андреевой Е.В., Гуровица В.М., Матюхина В.В. Московские олимпиады по информатике. М.: МЦНМО, 2006, 256 стр.
26. Семакин И.Г., Шестаков А.П. Основы алгоритмизации и программирования. М.: Академия, 2012, 400 стр.
27. Фиошин М.Е., Рессин А.А., Юнусов С.М. Информатика и ИКТ. В 2 ч. Ч. 2: 11 класс. М.: ДРОФА, 2008, 271 стр.

Acest manual este proprietatea Ministerului Educației al Republicii Moldova.

Liceul/gimnaziul _____				
Manualul nr. _____				
Nr. crt.	Numele de familie și prenumele elevului	Anul școlar	Aspectul manualului	
			la primire	la restituire
1.				
2.				
3.				
4.				
5.				

Dirigintele verifică dacă numele elevului este scris corect.

Elevul nu trebuie să facă niciun fel de însemnări în manual.

Aspectul manualului (la primire și la restituire) se va aprecia folosind termenii: *nou, bun, satisfăcător, nesatisfăcător*.

Imprimare la Tipografia „BALACRON” SRL,
str. Calea Ieșilor, 10; MD-2069
Chișinău, Republica Moldova
Comanda nr. 560