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 Șleahtițchi*, 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 „Alecu 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

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)

ISBN 978-9975-67-877-3

CUPRINS

Dragi prieteni,

Manualul este elaborat în conformitate cu Curriculumul disciplinar de informatică şi are drept scop însuşirea de către elevi a cunoştinţelor necesare pentru formarea culturii informaţionale şi dezvoltarea gîndirii algoritmice. Cu ajutorul acestui manual veţi studia funcţiile şi procedurile limbajului PASCAL, structurile dinamice de date şi metodele de elaborare a produselor program. De asemenea, veţi studia cele mai răspîndite tehnici de programare: trierea, tehnica Greedy, reluarea, metoda *desparte* şi *stăpîneşte*, programarea dinamică, metoda *ramifică* şi *mărgineşte*, algoritmii euristici. În manual sînt expuse metode de estimare a necesarului de memorie şi a timpului cerut de algoritmi, recomandări ce vizează utilizarea recursiei şi iterativităţii.

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

Implementarea tehnicilor de programare este ilustrată cu ajutorul mai multor probleme frecvent întîlnite în viaţa cotidiană şi studiate în cadrul disciplinelor şcolare din ciclul liceal. Totodată, în manual au fost incluse şi probleme de o reală importanţă practică, rezolvarea cărora este posibilă doar cu aplicarea calculatorului.

Fiind strîns legate de cunoştinţele din alte domenii, temele din manual sînt axate pe metodele de rezolvare a problemelor ce necesită un volum foarte mare de calcul, evidenţiindu-se rolul esenţial al gîndirii matematice în apariţia şi dezvoltarea informaticii. Exemplele, exerciţiile şi sarcinile individuale din manual vor contribui la perceperea adecvată a rolului şi locului calculatorului, a influenţei lui asupra dezvoltării matematicii, fizicii, chimiei, ştiinţelor socioumane. Pentru majoritatea temelor din manual au fost elaborate programe destinate instruirii asistate de calculator, fapt ce permite individualizarea procesului de predare–învăţare, organizarea lecţiilor practice şi dezvoltarea capacităţilor creative ale fiecărui elev.

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

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

Autorul

Capitolul 1

FUNCȚII ŞI 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. Cînd în programul principal se întîlnește un apel, execuția continuă cu prima instrucțiune din programul apelat (*fig. 1.1*). Cînd se termină executarea instrucțiunilor din subprogram, se revine la instrucțiunea imediat următoare apelului din programul principal.

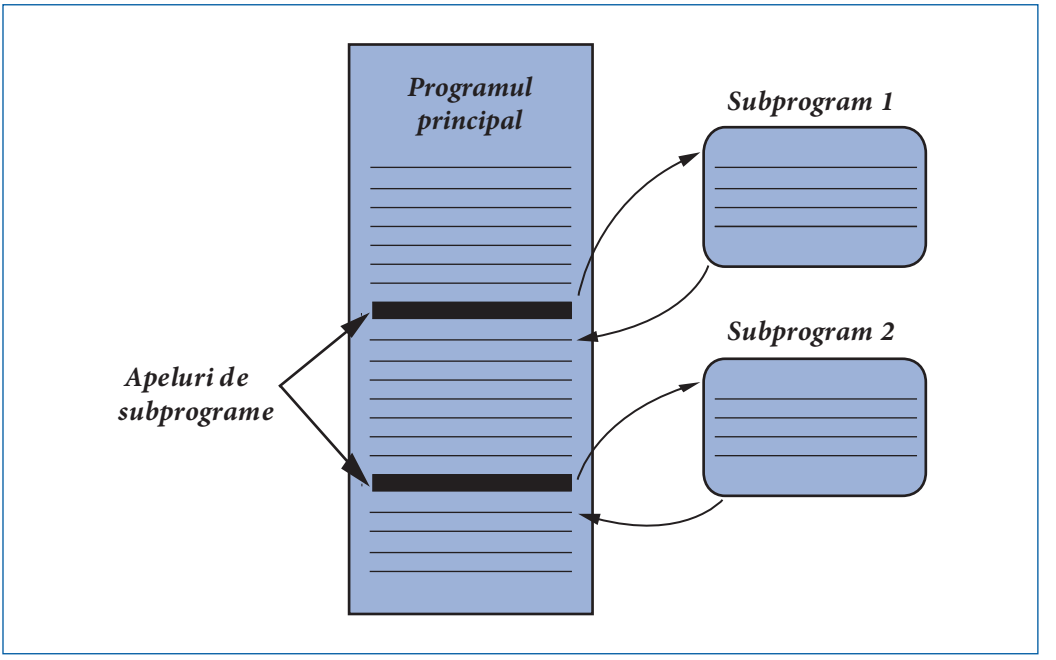

Fig. 1.1. Interacțiunea între program şi subprogram

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

<*Subprograme*> ::= { <*Funcție*>; | <*Procedură*>; }

Funcțiile sînt subprograme care calculează şi 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 acelaşi mod ca şi 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` ş.a., studiate în clasele precedente. În completare, programatorul poate defini proceduri proprii, care se apelează în acelaşi 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 şi proceduri se includ în partea executabilă a programului.

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

Întrebări şi exerciții

❶ Explicați termenii program principal şi subprogram.
❷ Cum interacționează programul şi subprogramul?
❸ Care este diferența dintre proceduri şi funcții?
❹ Cum se apelează o funcție? În care instrucțiuni ale limbajului pot apărea apeluri de funcții?
❺ Cum se apelează o procedură?
❻ Numiți tipul argumentului şi tipul rezultatului furnizat de funcțiile predefinite `abs`, `chr`, `eof`, `eoln`, `exp`, `ord`, `sin`, `sqr`, `sqrt`, `pred`, `succ`, `trunc`.
❼ Numiți tipul parametrilor actuali ai procedurilor `read` şi `write`.
❽ Ce prelucrări de date efectuează procedurile `read` şi `write`?

1.2. Funcții

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

```
function f(x1, x2, ..., xn) : tr;
D;
  begin
    ...
    f := e;
    ...
  end;
```

Prima linie este antetul funcției, format din:

f — numele funcției;

$(x_1, x_2, ..., x_n)$ — 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* şi 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 obişnuit, un parametru formal din lista (x_1, x_2, ..., x_n) are forma:

$$v_1, v_2, ..., v_k : t_p$$

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

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

$$f(a_1, a_2, ..., a_n)$$

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

Exemplu:

```
Program P97;
  {Declararea şi utilizarea funcţiei 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` şi `n` de tipul `Natural`. Funcția returnează o valoare de tipul `real`. În corpul funcției sînt declarate variabilele locale `p` şi `i`.

La execuția apelului `Putere(a,b)` valorile `3.0` şi `2` ale parametrilor actuali `a`, `b` se transmit parametrilor formali, respectiv, `x` şi `n`. De menționat că tipul lui `a` coincide cu tipul lui `x` şi 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` şi `n`. Totuşi apelul este corect, întrucît tipurile respectve sînt compatibile din punctul de vedere al atribuirii.

Întrebări şi exerciții

❶ 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 şi tipul rezultatului returnat de funcție. Precizați variabilele declarate în corpul funcției. Elaborați un program care afişează pe ecran valorile *n*! pentru *n* = 2, 3 şi 7.

❷ În care loc al programului principal se includ declarațiile de funcții?

❸ 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.
```

❹ 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')`

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

c) `F(4, 4, 4)`

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

e) `F(3.18, 4, 4)`

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

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

h) `F(15,21,3)`

❺ 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.

❻ 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.

❼ 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;
```

Elaboraţi o funcţie care calculează:
a) lungimea segmentului;
b) lungimea cercului;
c) aria cercului;
d) aria triunghiului;
e) aria dreptunghiului.

❽ Variabila *A* este introdusă prin declaraţia

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

Elaboraţi o funcţie care returnează numărul de caractere din mulţimea *A*.

❾ Elaboraţi o funcţie care să calculeze diferenţa în secunde între două momente de timp date prin oră, minute şi secunde.

❿ Un triunghi este definit prin coordonatele vîrfurilor sale. Scrieţi funcţii care, pentru două triunghiuri date, să studieze dacă:
a) au aceeaşi arie;
b) sînt asemenea;
c) primul este în interiorul celui de-al doilea.

1.3. Proceduri

Forma generală a textului unei declaraţii 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 opţională de parametri formali;
În **corpul procedurii** sînt incluse:
D — declaraţiile locale (opţionale) grupate după aceleaşi reguli ca şi în cazul funcţiilor;

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 introduşi prin declarații de forma

$v_1, v_2, ..., v_k : t_p$

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

Parametrii formali introduşi în listă prin declarații de forma

var $v_1, v_2, ..., v_k : t_p$

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

Activarea unei proceduri se face printr-un apel de forma

$p(a_1, a_2, ..., a_n)$

unde $a_1, a_2, ..., a_n$ 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 şi utilizarea procedurii Lac }
var   a, b, c, t, q : real;

procedure Lac(r : real; var l, s : real);
 {lungimea şi 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` şi `s`. Parametrul `r` este un parametru-valoare, iar `l` şi `s` sînt parametri-variabilă.

Execuţia instrucţiunii `Lac(a,b,c)` determină transmiterea valorii `1.0` drept valoare a parametrului formal `r` şi a locaţiilor (adreselor) variabilelor `b` şi `c` drept locaţii (adrese) ale parametrilor formali `l` şi `s`. Prin urmare, secvenţa de instrucţiuni

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

este echivalentă cu secvenţa

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

În mod similar, instrucţiunea

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

este echivalentă cu secvenţa

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

Întrebări şi exerciţii

❶ Care este diferenţa dintre un *parametru-valoare* şi un *parametru-variabilă*?

❷ Se consideră declaraţiile:

```
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)`

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

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

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

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

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

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

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

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

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

Argumentați răspunsul.

❸ 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.
```

❹ 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.

❺ Elaborați o procedură care:

a) calculează rădăcinile ecuației $ax^2 + bx + c = 0$;

b) radiază dintr-un șir caracterul indicat în apel;

c) 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 decît un număr natural dat *n*.

❻ 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 tînără;
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).

❼ 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.

❽ 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 şi 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 ataşează cîte 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 ş.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 şi acelaşi 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 şi se sfîrşeşte 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 acelaşi nume din blocul exterior.

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

Cunoaşterea 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`. Acelaşi 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 sfîrşitul antetului. Prin urmare, domeniul de vizibilitate al unei astfel de declarații include şi 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}`şi `{6}`. Domeniul de vizibilitate al declarației `d:integer` este textul cuprins între `{3}`şi `{5}`.

Întrebări şi exerciții

❶ Cum se determină domeniul de vizibilitate al unei declarații?

❷ Determinați domeniile de vizibilitate ale declarațiilor `b : real` şi `x : real` din programul `P105` (*fig. 1.2*).

❸ Precizați structura de bloc a programului ce urmează. Indicați domeniul de vizibilitate al fiecărei declarații şi determinați obiectele desemnate de fiecare apariție a identificatorilor `c` şi `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?

❹ Determinați domeniile de vizibilitate ale identificatorilor `P` și `F` din programul `P105` (*fig. 1.2*).

❺ 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.
```

❻ 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. Cunoaştem deja că datele de prelucrat şi 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 cînd ea este declarată în programul sau subprogramul ce îl cuprinde fără să fie redeclarată în subprogramul în studiu. Întrucît variabilele globale sînt cunoscute atît în subprogram, cît şi în afara lui, ele pot fi folosite pentru transmiterea datelor de prelucrat şi 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 afişează 2.0000000000E+00 }
  writeln(F);      {se afişează 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` şi 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 aceleaşi date. Pentru exemplificare amintim funcțiile cu argumente de tip *tablou*, procedurile care prelucrează tablouri şi fişiere de angajați, persoane, elevi etc.

Întrebări şi exerciții

❶ Explicați termenii *variabilă globală* relativ la un subprogram şi *variabilă locală* într-un subprogram.

❷ Numiți variabilele globale şi variabilele locale din programul `P105` (*fig. 1.2*).

❸ Poate fi oare o variabilă locală în acelaşi timp şi o variabilă globală relativ la un subprogram?

❹ Numiți variabilele globale şi variabilele locale din programul ce urmează. Ce va afişa 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.
```

❺ 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.

❻ 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 sfîrșit 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 obişnuit, 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 şi 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 şi 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 afişează 1 }
    writeln(F(1));  { se afişează 2 }
    writeln(F(1));  { se afişează 3 }
    readln;
end.
```

În programul `P110` funcţia `F` returnează valoarea expresiei `a*x`. Pe lîngă aceasta însă, atribuirea `a:=a+1` alterează valoarea variabilei globale `a`. În consecinţă, pentru una şi aceeaşi valoare `1` a argumentului `x` funcţia returnează rezultate diferite, fapt ce nu se încadrează în conceptul uzual de funcţie.

```
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 afişează 4 }
  writeln(F(a));     { se afişează 6 }
  writeln(F(a));     { se afişează 8 }
  readln;
 end.
```

În programul `P111` funcţia `F` returnează valoarea expresiei `2*x`. Întrucît `x` este un parametru formal variabilă, atribuirea `x:=x+1` schimbă valoarea parametrului actual din apel, şi anume a variabilei `a` din programul principal. Faptul că apelurile textual identice `F(a)`, `F(a)` şi `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 atribuiri la funcţii. 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 declaraţie şi parametri actuali în apel. Consecinţele efectelor colaterale se pot propaga în domeniul de vizibilitate al declaraţiilor globale şi pot interfera cu cele similare, produse la execuţia altor proceduri şi funcţii. În astfel de condiţii, 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 şi 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 şi 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 şi 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 declanşarea 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 şi oprirea procesului repetitiv; urmează revenirile din apeluri şi 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)` ş.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 însuşi: cînd execuția ajunge la acest apel, este declanşată o nouă execuție ş.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 opreşte cînd n ia valoarea 0; în cazul funcției `Fib` procesul se opreşte cînd n ia valoarea 0 sau 1.

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

— valorile curente ale parametrilor transmişi prin valoare;
— locaţiile (adresele) parametrilor-variabilă;
— adresa de retur, adică adresa instrucţiunii ce urmează după apel.

Prin urmare, la apeluri recursive spaţiul ocupat din memorie va creşte rapid, riscînd depăşirea capacităţii de memorare a calculatorului. Astfel de cazuri pot fi evitate, înlocuind recursia prin iteraţie (instrucţiunile **for**, **while**, **repeat**). Pentru exemplificare prezentăm o formă nerecursivă a funcţiei *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: translatarea programelor PASCAL în limbajul cod-maşină, grafica pe calculator, recunoaşterea formelor ş.a.

Întrebări şi exerciţii

❶ Cum se execută un subprogram recursiv? Ce informaţii se depun în memoria calculatorului la execuţia unui apel recursiv?

❷ Care este diferenţa dintre recursie şi iteraţie?

❸ Elaboraţi o formă nerecursivă a funcţiei lui *Fibonacci*.

❹ Scrieţi un subprogram recursiv care:
a) calculează suma $S(n) = 1 + 3 + 5 + ... + (2n - 1)$;
b) calculează produsul $P(n) = 1 \times 4 \times 7 \times ... \times (3n - 2)$;
c) inversează un şir de caractere;
d) calculează produsul $P(n) = 2 \times 4 \times 6 \times ... \times 2n$.

❺ Elaboraţi un program care citeşte de la tastatură numerele naturale *m, n* şi afişează pe ecran valoarea funcţiei lui *Ackermann*:

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

Calculaţi *a(0, 0)*, *a(1, 2)*, *a(2, 1)* şi *a(2, 2)*. Încercaţi să calculaţi *a(4, 4)* şi *a(10, 10)*. Explicaţi mesajele afişate pe ecran.

❻ Se consideră declaraţia

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

Elaboraţi un subprogram recursiv care:
a) afişează 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.

❼ Elaborați 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}
```

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

<*Număr*> ::= <*Cifră*> | <*Cifră*> <*Număr*>

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

<*Număr*> ::= <*Cifră*> {<*Cifră*>}

❾ Se consideră următoarele formule metalingvistice:

<*Număr*> ::= <*Cifră*> {<*Cifră*>}
<*Semn*> ::= + | –
<*Expresie*> ::= <*Număr*> | <*Expresie*> <*Semn*> <*Expresie*>

Scrieți o funcție recursivă care returnează valoarea `true` dacă șirul de caractere `s` este conform definiției unității lexicale <*Expresie*>.

1.8. Sintaxa declarațiilor şi apelurilor de subprograme

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

<*Funcție*> ::= <*Antet funcție*>; <*Corp*>
| <*Antet funcție*>; <*Directivă*>
| **function** <*Identificator*>; <*Corp*>

<*Antet funcție*> ::=
function <*Identificator*> [<*Listă parametri formali*>] **:** <*Identificator*>

Diagramele sintactice sînt prezentate în *figura 1.3*.

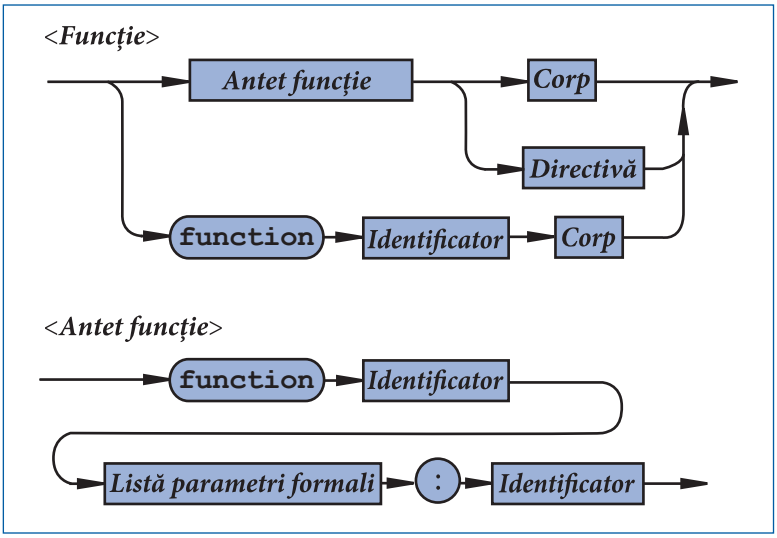

Fig. 1.3. Sintaxa declarațiilor de funcții

Procedurile se definesc cu ajutorul următoarelor formule:

<*Procedură*>::=<*Antet procedură*>**;**<*Corp*> | <*Antet procedură*>**;**<*Directivă*>
| **procedure** <*Identificator*>**;**<*Corp*>

<*Antet procedură*> := **procedure** <*Identificator*> [<*Listă parametri formali*>]

Diagramele sintactice sînt prezentate în *figura 1.4*.

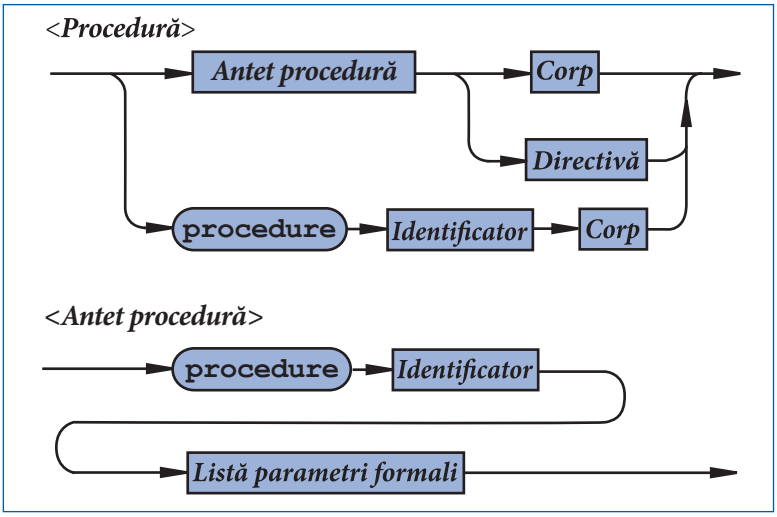

Fig. 1.4. Sintaxa declarațiilor de proceduri

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*.

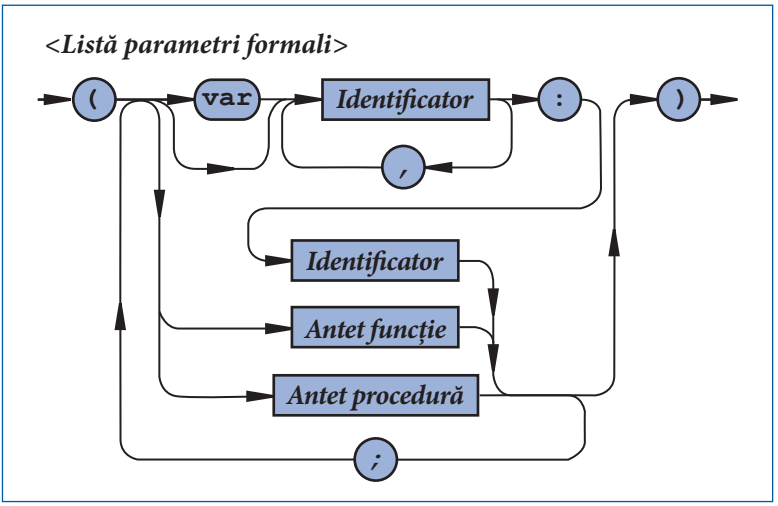

Fig. 1.5. Diagrama sintactică <*Listă parametri formali*>

Amintim că în lipsa cuvîntului-cheie **var** identificatorii din listă specifică **parametrii-valoare**. Cuvîntul **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 aparținînd unui **tip procedural** şi au forma parametrilor-valoare. Limbajul PASCAL extinde sensul uzual al noțiunii de funcție, permițînd returnarea valorilor nu numai prin numele funcției, ci şi 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*.

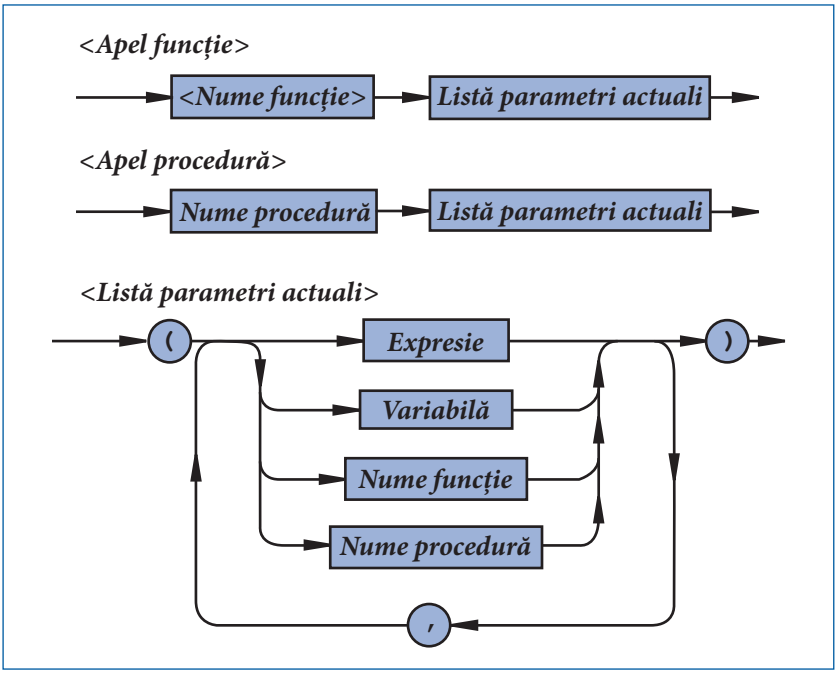

Fig. 1.6. Sintaxa apelurilor de funcții și proceduri

Corespondența între un parametru actual şi parametrul formal se face prin poziția ocupată de aceştia 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 şi exerciții

❶ Cînd 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.

Capitolul 2

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 stabileşte în momentul scrierii programului şi 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 şir de aşteptare (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 şi 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 defineşte printr-o declarație de forma:

```
type Tr = ^Tb;
```

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

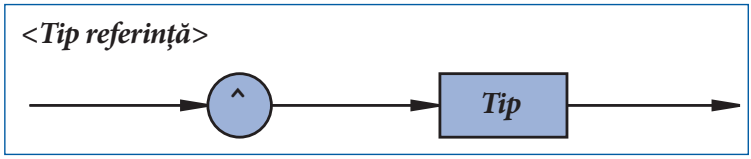

Fig. 2.1. Diagrama sintactică <*Tip referință*>

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` şi `c` identifică variabile de tipul `real` şi, respectiv, `char`. Subliniem faptul că tipurile de date `AdresaInteger`, `AdresaChar` şi tipul anonim `^real` sînt tipuri *referință* distincte.

Operațiile care se pot face cu valorile unui tip de date *referință* sînt = şi <>. Valorile de tip *referință* nu pot fi citite de la tastatură şi afişate 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 şi returnează adresa zonei respective prin variabila `p`. În continuare variabila dinamică poate fi accesată prin aşa-zisa **dereperare**: numele variabilei de tip *referință* `p` este urmat de semnul de „`^`”. Dereperarea unei variabile de tip *referință* cu conținutul **`nil`** va declanşa 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 şir $v_1, v_2, ..., v_n$ 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 şi 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;
 {Operaţii 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);
  {operaţii cu variabilele create }
  i^:=1; j^:=2;
  k^:=i^+j^;
  writeln(k^);
  {crearea variabilelor dinamice de tipul real }
  new(r); new(s); new(t);
  {operaţii 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` şi pot fi reutilizate pentru crearea unor variabile dinamice noi. Prin urmare, procedurile `new` şi `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 şi spațiul de memorie disponibil. În cazul în care tot spațiul de memorie este deja ocupat, apelul procedurii `new` va declanşa o eroare de execuție.

Exemplu:

```
Program P118;
 {Eroare: depăşirea capacităţii memoriei }
 label 1;
 var i : ^integer;
```

```
begin
 1 : new(i);
     goto 1;
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 Comentaţi 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 Explicaţi expresia *alocarea dinamică a memoriei*.

2.2. Structuri de date

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

Tablourile, şirurile de caractere, articolele, fişierele şi mulţimile studiate în capitolele precedente sînt **structuri implicite de date**. Relaţiile dintre componentele acestor sructuri sînt predefinite şi nemodificabile. De exemplu, toate componentele unui şir de caractere au un nume comun, iar caracterul `s[i+1]` este succesorul caracterului `s[i]` în virtutea poziţiei ocupate.

Întrucît structura tablourilor, şirurilor de caractere, articolelor, mulţimilor şi fişierelor nu se modifică în timpul execuţiei 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 relaţiile dintre componente nu numai că se modifică dinamic, dar în acelaşi timp pot deveni deosebit de complexe.

De exemplu, în cazul unui fir de aşteptare la o casă de bilete relaţiile dintre persoane se modifică: persoanele nou-sosite se aşază la rînd; persoanele în criză de timp pleacă fară să-şi mai procure bilete; persoanele care au plecat pentru un timp îşi păstrează rîndul ş.a.m.d. În cazul proiectării asistate de calculator a reţelelor de circula-

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

Prin urmare, este necesară folosirea unor structuri de date în care relațiile dintre componente să fie reprezentate şi prelucrate în mod explicit. Acest efect se poate obține ataşînd 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 şi eventual distruse în timpul execuției programului se numesc **structuri dinamice de date**. Structurile dinamice frecvent utilizate sînt: listele unidirecționale, listele bidirecționale, stivele, cozile, arborii ş.a.

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

Întrebări şi exerciții

❶ Explicați termenul *structură de date*. Dați exemple.

❷ Care este diferența dintre structurile implicite şi structurile explicite de date?

❸ O structură de date este **omogenă** dacă toate componentele sînt de acelaşi tip. În caz contrar, structura de date este **eterogenă**. Dați exemple de structuri omogene și structuri eterogene de date.

❹ Care este diferența dintre structurile statice şi structurile dinamice de date?

❺ Explicați termenul *structură recursivă de date*.

2.3. Liste unidirecționale

Listele unidirecționale sînt structuri explicite şi dinamice de date formate din celule. Fiecare **celulă** este o variabilă dinamică de tipul **`record`** ce conține, în principal, două cîmpuri: cîmpul datelor şi 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* şi *D*.

Datele necesare pentru crearea şi 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 cîmpul `Info`, iar adresa celulei următoare în cîmpul `Urm`. Pentru simplificare se consideră că cîmpul `Info` este de tipul *șir de caractere*. Ultima celulă din listă va avea în cîmpul `Urm` valoarea **`nil`**. Adresa primei celule (adresa de bază) este memorată în variabila de tip referință `P` (*fig. 2.2*).

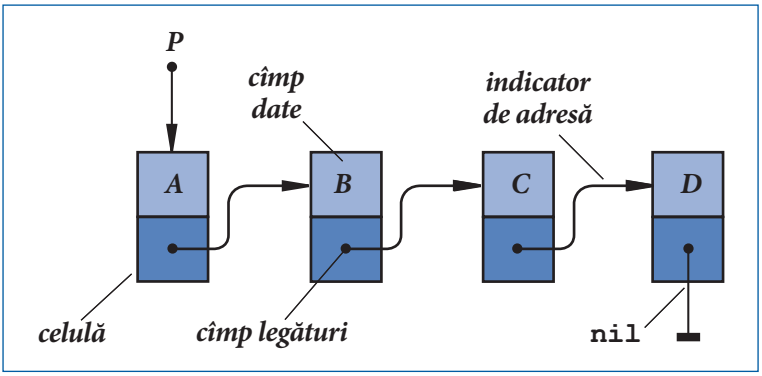

Fig. 2.2. Lista unidirecțională

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 tîrziu în aceeași declarație.

O listă unidirecțională poate fi creată adăugînd la vîrful listei cîte 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 informaţiei utile }
  R^.Urm:=nil;  {înscrierea indicatorului "sfîrşit de listă" }
  V:=R;         {memorarea adresei vîrfului }
  {3 - adăugarea celulei B }
  new(R);       {crearea unei celule }
  R^.Info:='B'; {încărcarea informaţiei utile }
  R^.Urm:=nil;  {înscrierea indicatorului "sfîrşit de listă" }
  V^.Urm:=R;    {crearea legăturii A -> B }
  V:=R;         {actualizarea adresei vîrfului }
  {4 - adăugarea celulei C }
  new(R);       {crearea unei celule }
  R^.Info:='C'; {încărcarea informaţiei utile }
  R^.Urm:=nil;  {înscrierea indicatorului "sfîrşit de listă" }
  V^.Urm:=R;    {crearea legăturii B -> C }
  V:=R;         {actualizarea adresei vîrfului }
  {5 - adăugarea celulei D }
  new(R);       {crearea unei celule }
  R^.Info:='D'; {încărcarea informaţiei utile }
  R^.Urm:=nil;  {înscrierea indicatorului "sfîrşit de listă" }
  V^.Urm:=R;    {crearea legăturii C -> D }
  V:=R;         {actualizarea adresei vîrfului }
  {afişarea 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 vîrfului) din programul `P122` reţine adresa ultimei celule deja create pentru a-i poziţiona indicatorul de adresă `V^.Urm`. Se procedează astfel pentru că în momentul în care completăm cîmpurile `R^.Info` şi `R^.Urm` ale celulei curente încă nu se cunoaşte adresa celulei ce urmează.

Listele cu un număr arbitrar de celule pot fi create şi afişate cu ajutorul procedurilor respective din programul `P123`:

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

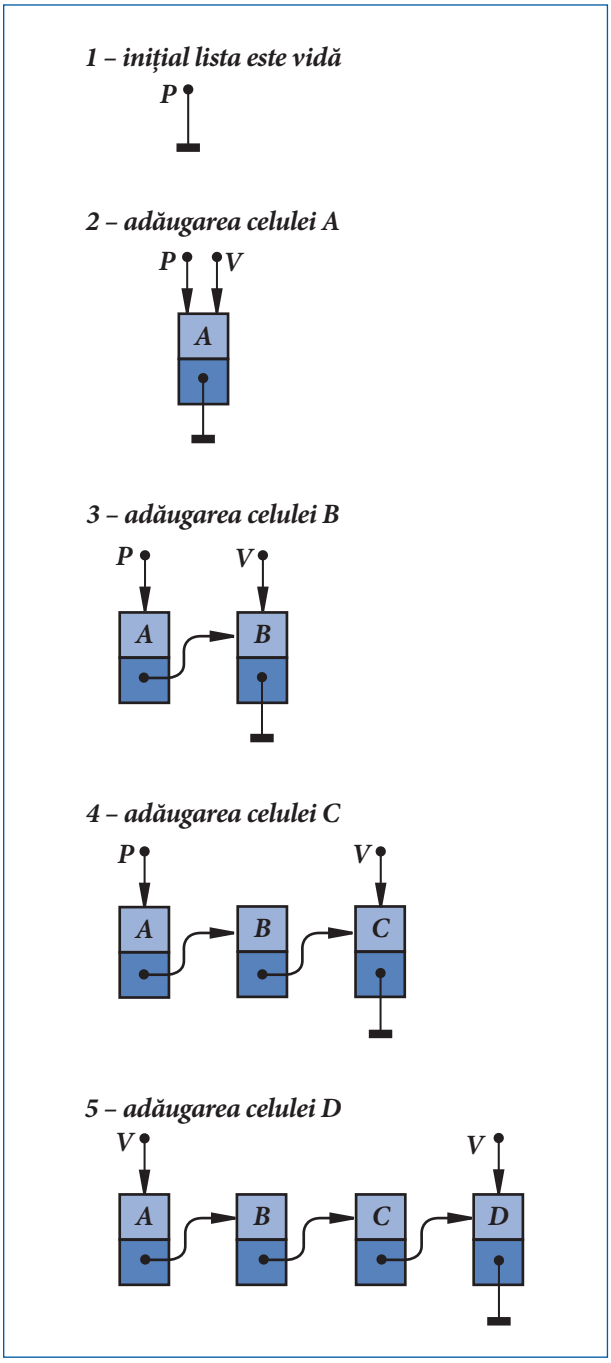

Fig. 2.3. Crearea listei unidirecționale

```
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 şi prelucrarea structurilor de date cu un număr arbitrar de elemente.

Întrebări şi exerciții

❶ Cum se definesc datele necesare pentru crearea unei liste?

❷ Care este destinația cîmpului datelor din componența unei celule? Care este destinația cîmpului legăturilor? Ce informație se înscrie în acest cîmp?

❸ Scrieți un program care creează lista unidirecțională din *figura 2.2*, adăugînd cîte un element la baza listei.

❹ Elaborați o procedură care schimbă cu locul două elemente din listă.

❺ 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.

❻ 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 şi 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 (ştergerea) unui element dintr-o listă ş.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 cîmpul 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 vîrful 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: ...
```

Întrucît listele unidirecționale sînt structuri recursive de date, operația de căutare poate fi realizată şi 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;
```

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:

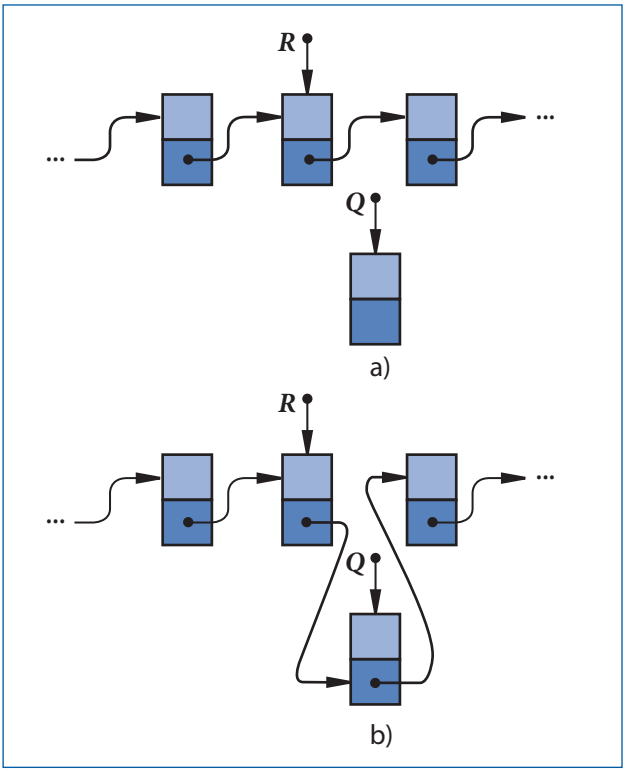

Fig. 2.4. Includerea unui element în listă:
a – lista pînă la includere; *b* – lista după includere

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

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

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

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

Exemplu:

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

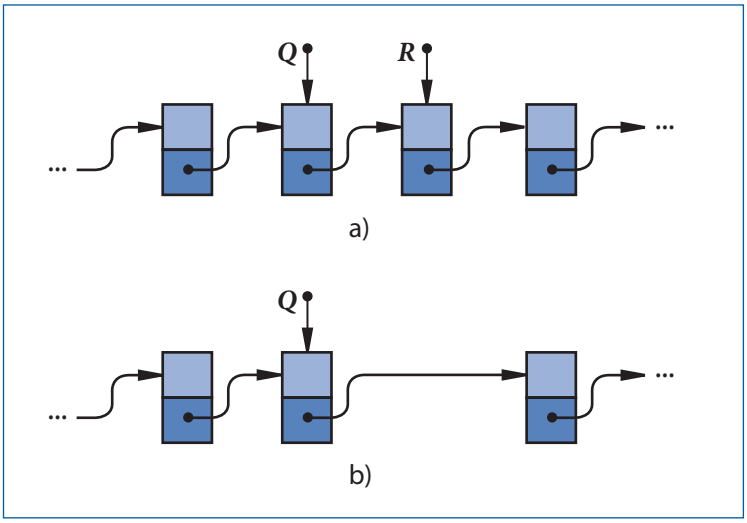

Fig. 2.5. Excluderea unui element din listă:
a – lista pînă la excludere; *b* – lista după excludere

```
 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 citeşte de la tastatură. Procedura `Afis` afişează elementele listei la ecran. Procedura `Includ` citeşte de la tastatură elementul ce urmează să fie inclus şi 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 şi elimină, dacă există, celula ce conține elementul citit de la tastatură.

Întrebări şi exerciții

❶ Scrieți o funcție nerecursivă care returnează adresa vîrfului listei unidirecționale. Transcrieți această funcție într-o formă recursivă.

❷ Transcrieți procedurile `Includ` şi `Exclud` din programul `P124`, fără a utiliza instrucțiunea **`goto`**.

❸ Se consideră următoarele tipuri de date:

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

Elaborați un program care:
a) creează o listă unidirecțională cu componente de tipul `Candidat`;
b) afișează lista pe ecran;
c) exclude din listă candidatul care își retrage actele;
d) include în listă candidatul care depune actele;
e) afișează pe ecran candidații cu media mai mare de 7,5;
f) creează o listă suplimentară formată din candidații cu media mai mare de 9,0;
g) exclude din listă toți candidații cu media mai mică de 6,0.

❹ Elaborați o procedură care:
a) reordonează elementele listei unidirecționale conform unui anumit criteriu;
b) concatenează două liste unidirecționale;
c) descompune o listă în două liste;
d) selectează din listă elementele care corespund unui anumit criteriu.

❺ Elementele listei unidirecționale sînt memorate într-un tablou unidirecțional. Elaborați procedurile necesare pentru:
a) parcurgerea listei;
b) căutarea unui anumit element;
c) includerea unui element;
d) excluderea unui element.

Care sînt avantajele și neajunsurile acestei reprezentări? Se consideră că lista va conține cel mult 100 de elemente.

❻ Scrieți o funcție recursivă ce returnează numărul de celule dintr-o listă unidirecțională.

❼ Scrieți un subprogram recursiv care exclude o anumită celulă din listă.

❽ 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:
a) cuvintele se includ în listă în ordinea primei lor apariții în text;
b) 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 numeşte **stivă vidă**.

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

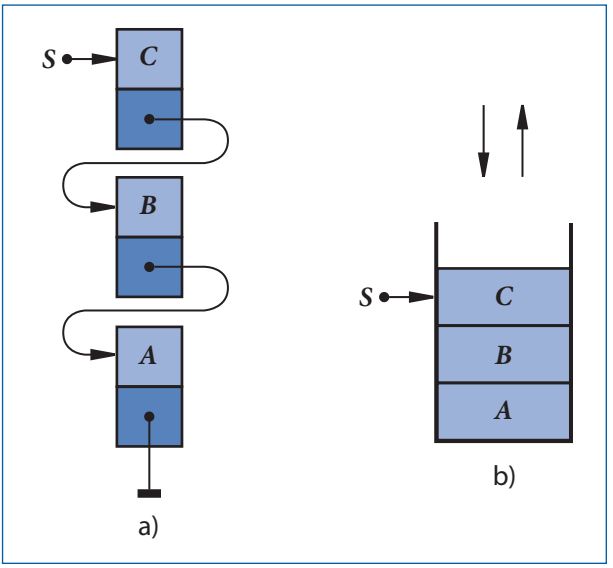

Fig. 2.6. Stiva:
a – reprezentarea detaliată; *b* – reprezentarea generalizată

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

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

Adresa vîrfului stivei este memorată în variabila de tip *referință* `S`. Adresa celulei precedente din stivă este memorată în cîmpul `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 cîmpul R^.Info }
R^.Prec:=S; { crearea legăturii către celula precedentă
              din stivă }
S:=R; { actualizarea adresei vîrfului }
```

unde `R` este o variabilă de tipul `AdresaCelula`.

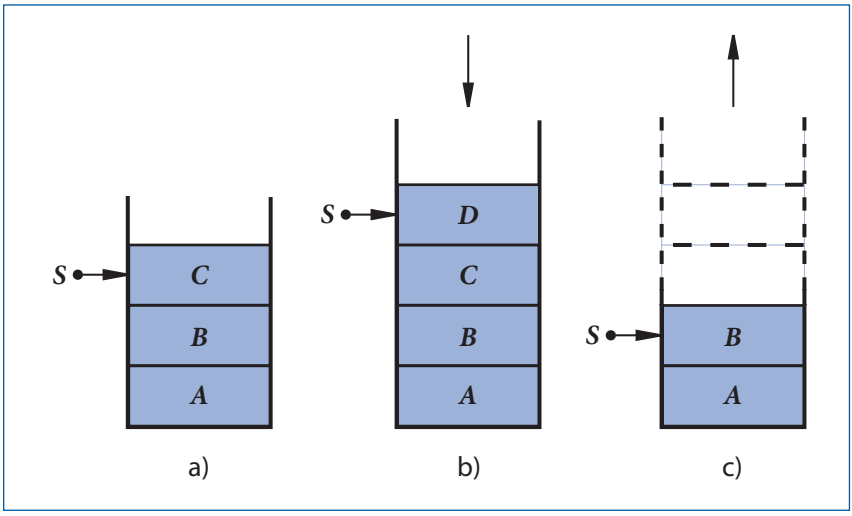

Fig. 2.7. Introducerea și extragerea elementelor din stivă:
a – stiva inițială; *b* – introducerea elementului *D*; *c* – extragerea elementelor *D*, *C*

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

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

Exemplu:

```
Program P127;
 {Crearea şi prelucrarea unei stive }
 type AdresaCelula=^Celula;
      Celula=record
                   Info : string;
                   Prec : AdresaCelula;
             end;
 var S : AdresaCelula; {adresa vîrfului }
     c : char;
 procedure Introduc;
 var R : AdresaCelula;
 begin
  new(R);
  write('Daţi 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 şi numele de **liste *LIFO*** (*last in, first out* — ultimul element care a intrat în stivă va fi primul care va ieşi din ea) şi sînt frecvent utilizate pentru alocarea dinamică a memoriei în cazul procedurilor şi funcţiilor 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 şi exerciţii

❶ Care este ordinea de introducere şi extragere a datelor din stivă?

❷ De la tastatură se citesc mai multe numere naturale. Afişaţi numerele în studiu pe ecran în ordinea inversă citirii.

❸ În *figura 2.8* este reprezentată schema de manevrare a vagoanelor de tren într-un depou. Elaboraţi un program care citeşte de la tastatură şi afişează pe ecran datele despre fiecare vagon intrat sau ieşit din depou. Datele în studiu includ:
– numărul de înmatriculare (`integer`);
– staţia de înmatriculare (**`string`**);
– anul fabricării (`1960..2000`);
– tipul vagonului (**`string`**);
– capacitatea de încărcare (`real`);
– proprietarul vagonului (**`string`**).

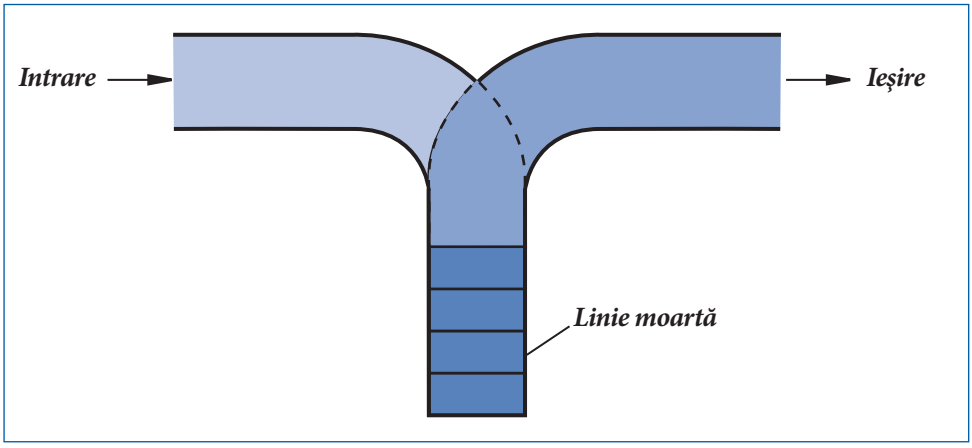

Fig. 2.8. Schema de manevrare a vagoanelor de tren

❹ Colectivele temporare de muncă sînt formate şi desfiinţate în ordinea „ultimul angajat va fi primul care va fi concediat". Elaboraţi un program care citeşte de la tastatură şi afişează pe ecran datele despre fiecare persoană angajată sau concediată. Datele în studiu includ:
– numele (**`string`**);
– prenumele (**`string`**);
– anul naşterii (`1930..1985`);
– data angajării (ziua, luna, anul).

❺ Se consideră şiruri finite de caractere formate din parantezele (,), [,], {, }. Un şir este **corect** numai atunci cînd el poate fi construit cu ajutorul următoarelor reguli:
a) şirul vid este corect;

b) dacă *A* este un şir corect, atunci (*A*), [*A*] şi {*A*} sînt şiruri corecte;

c) dacă *A* şi *B* sînt şiruri corecte, atunci *AB* este un şir corect.

De exemplu, şirurile ( ), [ ], { }, [( )], ((({[ ]}))([ ])) sînt corecte, iar şirurile ([, ( )[ ]{{, ([)] nu sînt corecte. Elaboraţi un program care verifică dacă şirul citit de la tastatură este corect.

Indicaţie. Problema poate fi rezolvată printr-o singură parcurgere a şirului supus verificării. Dacă caracterul curent este (, [sau {, el este depus în stivă. Dacă vîrful stivei şi caracterul curent formează una din perechile ( ), [ ] sau { }, paranteza respectivă este scoasă din stivă. În cazul unui şir corect, după examinarea ultimului caracter din şir stiva rămîne vidă.

❻ Elementele stivei sînt memorate într-un tablou unidimensional. Elaboraţi procedurile necesare pentru introducerea şi extragerea elementelor din stivă. Care sînt avantajele şi neajunsurile acestei reprezentări? Se consideră că stiva conţine cel mult 100 de elemente.

2.6. Cozi

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

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

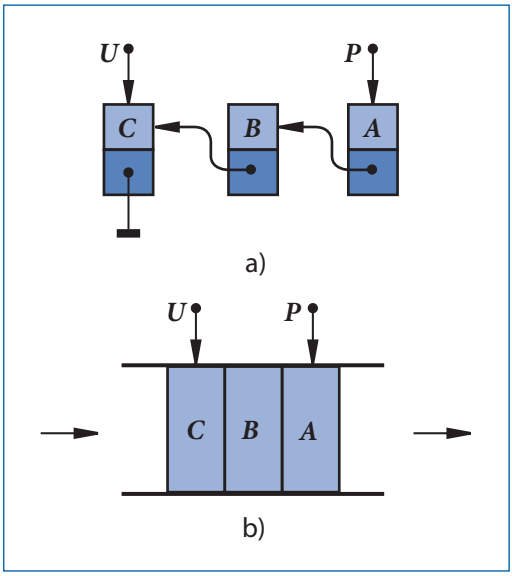

Fig. 2.9. Coada:
a – reprezentarea detaliată; *b* – reprezentarea generalizată

Datele necesare pentru crearea şi prelucrarea unei cozi pot fi definite prin declaraţii 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 cîmpul `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 cîmpul R^.Info }

R^.Urm:=nil; { înscrierea indicatorului "ultimul element" }
U^.Urm:=R;   { adăugarea celulei la coadă }
U:=R;        { actualizarea adresei ultimei celule }
```

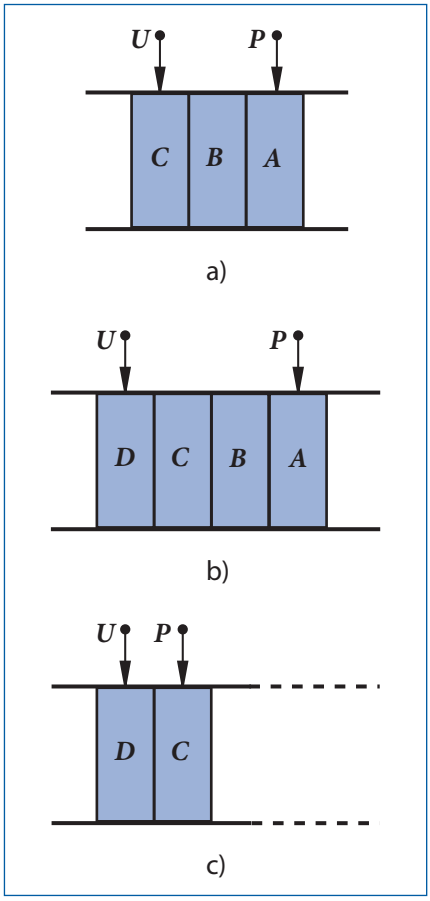

Fig. 2.10. Introducerea și extragerea elementelor din coadă:
a – coada inițială; *b* – introducerea elementului D;
c – extragerea elementelor A, B

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

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

Exemplu:

```
Program P128;
 {Crearea şi 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('Coada 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('Coada este vidă')
          else begin
                write('Coada include');
                writeln(' elementele:');
                R:=P;
                while R<>nil do
                 begin
                   writeln(R^.Info);
                   R:=R^.Urm;
                 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 - Afişarea 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 ieşit din coadă). Menționăm că simularea cozilor cu ajutorul tablourilor unidimensionale este ineficientă din cauza migrării elementelor cozii spre ultima componentă a tabloului.

Întrebări şi 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 şi avionul care aterizează. Datele în studiu includ:
– numărul de înmatriculare (`integer`);
– tipul avionului (**`string`**);
– numărul rutei (`integer`).

❸ Prin **coadă cu priorităţi** vom înţelege o coadă în care elementul de introdus se inserează nu după ultimul element al cozii, ci înaintea tuturor elementelor cu o prioritate mai mică. Priorităţile elementelor se indică prin numere întregi. Elaboraţi un program care:
a) creează o coadă cu priorităţi;
b) introduce în coadă elementele specificate de utilizator;
c) extrage elementele din coadă;
d) afişează coada cu priorităţi pe ecran.

2.7. Arbori binari

Prin **nod** se înţelege o variabilă dinamică de tipul **`record`** care conţine un cîmp destinat memorării informaţiilor utile şi doi indicatori de adresă.

Arborele binar se defineşte recursiv după cum urmează:
a) un nod este un arbore binar;
b) un nod ce conţine legături către alţi doi arbori binari este un arbore binar.

Prin convenţie, **arborele vid** nu conţine niciun nod. Pentru exemplificare, în *figura 2.11* este prezentat un arbore binar nodurile căruia conţin informaţia utilă *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*. Datele necesare pentru crearea şi prelucrarea unui arbore binar pot fi definite prin declaraţii 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, declaraţiile 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 numeşte **rădăcină**. Adresa rădăcinii se memorează în variabila de tip *referinţă* `T`. În cazul unui arbore vid `T=nil`.

Cei doi arbori conectați la rădăcină se numesc **subarborele stîng** şi **subarborele drept**. Adresa subarborelui stîng se memorează în cîmpul `Stg`, iar adresa subarborelui drept — în cîmpul `Dr`.

Nivelul unui nod este, prin convenție, 0 pentru nodul-rădăcină şi $i + 1$, pentru nodul conectat la un nod de nivelul i. În mod obişnuit, 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 ş.a.m.d. (*fig. 2.11*).

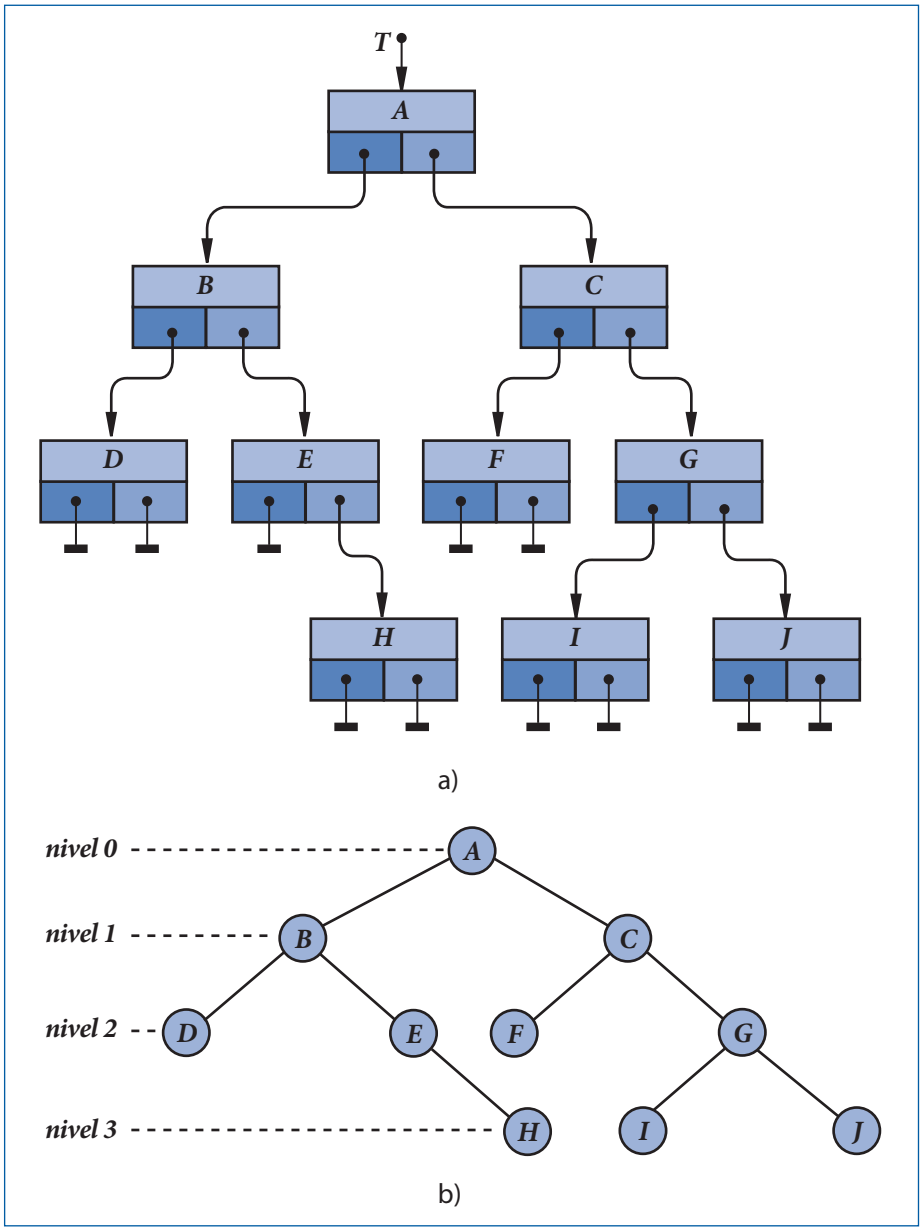

Fig. 2.11. Arborele binar:
a – reprezentarea detaliată; *b* – reprezentarea generalizată

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* ş.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* şi *C*; nodul *B* este părintele nodurilor *D* şi *E* ş.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* şi *J* sînt noduri terminale; nodurile *A*, *B*, *C*, *E* şi *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 şi 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 şi afişarea nodurilor respective pe ecran:

– se creează o coadă care conține un singur element – nodul-rădăcină;

– fiecare nod extras din coadă este afişat pe ecran;

– descendenții nodului extras se introduc în coadă;

– procesul de afişare 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('Coada este vidă')
          else begin
                 R:=Prim;
                 Q:=R^.Info;
                 Prim:=Prim^.Urm;
                 dispose(R);
               end;
end; { ExtrageDinCoada }
procedure CreareArboreBinar;
var R, Q : AdresaNod;
       s : string;
begin
 T:=nil;                {iniţial arborele este vid }
 Prim:=nil; Ultim:=nil; {iniţial coada este vidă }
 writeln('Daţi rădăcina:'); readln(s);
 if s<>'' then
     begin
     new(R); {crearea rădăcinii }
     R^.Info:=s;
     T:=R;   {iniţializarea adresei rădăcinii }
     IntroduInCoada(T);
    end;
 while Prim<>nil do  {cît coada nu e vidă }
    begin
     ExtrageDinCoada(R);
     writeln('Daţi descendenţii nodului', R^.Info);
     write('  stîng:  '); readln(s);
     if s='' then R^.Stg:=nil
        else
            begin
             new(Q); R^.Stg:=Q;
             Q^.Info:=s;
             IntroduInCoada(Q);
            end; { else }
```

```
     write('  drept:  '); readln(s);
      if s='' then R^.Dr:=nil
         else
             begin
              new(Q); R^.Dr:=Q;
              Q^.Info:=s;
              IntroduInCoada(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;
       IntroduInCoada(T);
       while Prim<>nil do
             begin
              ExtrageDinCoada(R);
              writeln('Nodul  ', R^.Info);
              write('  descendenți: ');
              if R^.Stg=nil then write('nil, ')
                 else begin
                       write(R^.Stg^.Info, ', ');
                       IntroduInCoada(R^.Stg);
                      end;

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

begin
 CreareArboreBinar;
 AfisareArboreBinar;
end.
```

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

tatură un şir vid de caractere). Menţionăm că coada creată de programul `P129` nu conţine nodurile propriu-zise, ci adresele acestor noduri.

Algoritmul recursiv construieşte arborii binari urmînd direct definiţia respectivă:

– se creează nodul-rădăcină;
– se construieşte subarborele stîng;
– se construieşte 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 cîmpul `Info` și returnează adresa nodului. În momentul cînd trebuie completate cîmpurile `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 subarborele stîng, rădăcina și apoi subarborele 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 translatoarele limbajelor de programare.

Întrebări și exerciții

❶ Cum se definește un *arbore binar*? Explicați termenii: *rădăcină, subarborele stîng, subarborele drept, descendent, nivel, nod terminal, nod neterminal, înălțimea arborelui binar*.

❷ Formulați algoritmii iterativi destinați creării și afișării arborilor binari.

❸ Cum se construiește un arbore binar cu ajutorul algoritmului recursiv?

❹ 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.

❺ Cum trebuie modificată procedura `AfisArb` din programul `P130` pentru ca arborele binar să fie afișat în ordinea: subarborele drept, nodul-rădăcină, subarborele stîng?

❻ Scrieți o funcție recursivă care returnează numărul nodurilor unui arbore binar. Transcrieți această funcție într-o formă nerecursivă.

❼ 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 cîştigătorul meciului la care au participat cei doi jucători din nodurile descendente. Evident, rădăcina arborelui va conține datele despre cîştigătorul turneului.
Scrieți un program care creează în memoria calculatorului şi afişează pe ecran arborele unui turneu prin eliminare.
Indicație: Se porneşte de la o listă a jucătorilor. Cîştigătorii meciurilor din prima etapă se includ într-o altă listă. În continuare se formează lista cîştigătorilor meciurilor din etapa a doua ş.a.m.d.

❽ 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*?

❾ 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 vîrful 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 şi 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.

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

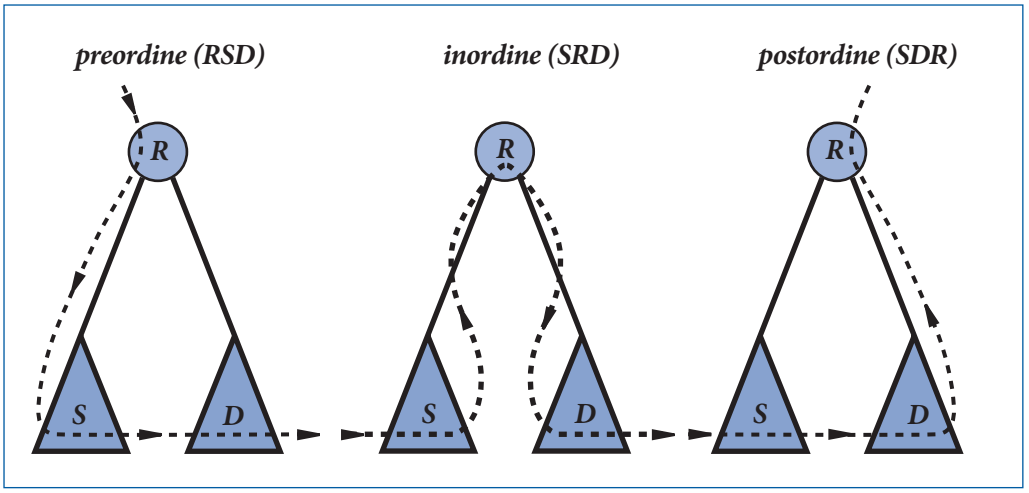

Fig. 2.12. Metodele de parcurgere a arborilor binari

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, parcurgînd arborele binar în curs de construcție în preordine. Procedura `AfisArb` afișează nodurile, parcurgînd arborele binar în inordine.

Întrebări şi 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.

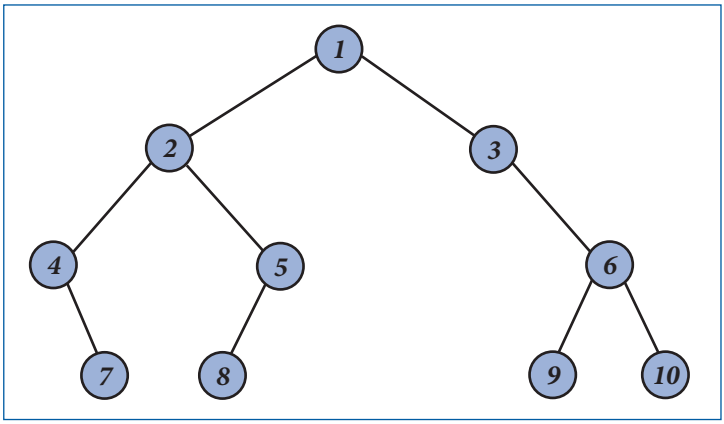

Fig. 2.13. Arborele binar

❻ Elaborați un program care afișează pe ecran toate nodurile aflate pe un nivel dat într-un arbore binar.

❼ Elaborați o procedură recursivă care parcurge un arbore binar în ordinea:
a) RDS (rădăcina – subarborele drept – subarborele stîng);
b) DRS (subarborele drept – rădăcina – subarborele stîng);
c) DSR (subarborele drept – subarborele stîng – rădăcina).
Transcrieți procedura elaborată într-o formă nerecursivă.

❽ Scrieți un subprogram care afișează la ecran nivelul fiecărui nod dintr-un arbore binar.

❾ Se dă un arbore binar în care nodurile terminale reprezintă numere întregi, iar cele neterminale – operațiile binare +, –, *, **mod**, **div**. Arborele în studiu poate fi considerat ca o reprezentare a unei expresii aritmetice. Valoarea acestei expresii se calculează efectuînd operația din nodul-rădăcină asupra valorilor subexpresiilor reprezentate de subarborele stîng și subarborele drept.
Scrieți o funcție care returnează valoarea expresiilor aritmetice reprezentate prin arbori binari.

❿ Se consideră expresiile aritmetice formate din operanzi și operatorii binari +, –, *, /. Operanzii sînt variabile numele cărora este format dintr-o singură literă și constante alcătuite dintr-o cifră. Fiecărei expresii aritmetice i se poate asocia un arbore binar după cum urmează:
a) expresiei aritmetice formate dintr-un singur operand i se asociază un arbore binar format doar din nodul ce conține operandul respectiv;
b) expresiei aritmetice de forma $E_1 \circ E_2$, unde E_1 și E_2 sînt expresii aritmetice, i se asociază un arbore binar care are în nodul-rădăcină operatorul $\circ$, ca subarbore stîng arborele asociat expresiei E_1, iar ca subarbore drept arborele asociat expresiei E_2.
Valoarea expresiei se calculează efectuînd operația din nodul-rădăcină asupra valorilor subexpresiilor reprezentate de subarborele stîng și subarborele drept.
Scrieți un program care:
a) construiește arbori binari asociați expresiilor aritmetice citite de la tastatură;
b) evaluează expresiile aritmetice reprezentate prin arborii binari.

Indicație. Algoritmul va urma definiția recursivă a arborelui în studiu. Ca operator curent „$\circ$" se poate desemna orice operator +, – din expresia supusă prelucrării. Operatorii *, / pot fi desemnați ca operatori curenți numai cînd expresia supusă prelucrării nu conține operatorii +, –.

2.9. Arbori de ordinul *m*

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

Arborele de ordinul *m* se defineşte 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** şi au fost studiați în paragrafele precedente. Arborii de ordinul 3, 4, 5 ş.a.m.d. se numesc **arbori multicăi** (în engleză *multiway 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 aceeaşi semnificație ca şi 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.

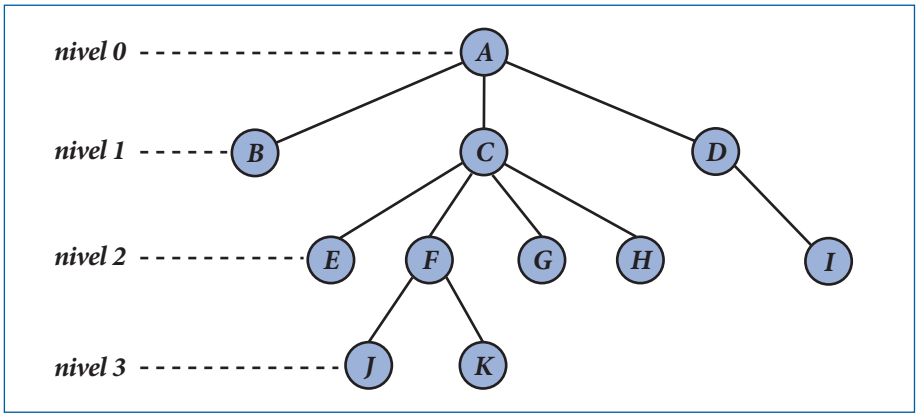

Fig. 2.14. Arborele de ordinul 4

Datele necesare pentru crearea şi 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 şi 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 obişnuit, parcurgerea în lățime se realizează cu ajutorul unui algoritm iterativ care utilizează o structură auxiliară de date, şi anume o coadă formată din adresele nodurilor care vor fi vizitate.

Parcurgerea în adîncime se defineşte recursiv: dacă arborele este vid, el este parcurs fără a se face nimic; altfel se vizitează întîi 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 IntroduInCoada(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('Coada este vidă')
           else begin                    R:=Prim;
                  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 }
       IntroduInCoada(T);
     end;
  while Prim<>nil do  {cît coada nu e vidă }
     begin
       ExtrageDinCoada(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;
           IntroduInCoada(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;
        IntroduInCoada(T);
        while Prim<>nil do
           begin
             ExtrageDinCoada(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, ' ');
                     IntroduInCoada(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;
      IntroduInCoada(T);
      while Prim<>nil do
        begin
          ExtrageDinCoada(R);
          writeln(R^.Info);
          for i:=1 to m do
           if R^.Dsc [i]<>nil then IntroduInCoada(R^.Dsc [i]);
        end; {while }
      end; {then }
end; {InLatime }

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 adîncime:');
 InAdincime(T);
 readln;
end.
```

Informația utilă asociată fiecărui nod se citeşte de la tastatură. Absența descendenților se semnalează prin apăsarea tastei <*ENTER*>. Menționăm că procedura `CreareArbore` creează nodurile parcurgînd 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 ş.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 şi 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 şi exerciții

❶ Dați exemple de arbori de ordinul 3, 5, 6.

❷ Cum se definește un arbore de ordinul *m*? Ce operații pot fi efectuate asupra arborilor în studiu?

❸ Explicați metodele de parcurgere a arborilor multicăi. Dați exemple.

❹ Scrieți un program recursiv care construiește în memoria calculatorului un arbore multicăi. Informația utilă asociată nodurilor se citește de la tastatură.

❺ Scrieți o funcție care returnează:
a) numărul nodurilor unui arbore multicăi;
b) nivelul unui anumit nod din arbore;
c) înălțimea arborelui.

❻ Transcrieți procedura `InAdincime` din programul `P133` într-o formă nerecursivă.

❼ 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`?

❽ 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*?

❾ Se dă un arbore multicăi, informațiile din noduri fiind şiruri de caractere. Să se afişeze pe ecran toate şirurile de caractere de lungime pară.

⑩ Organizarea datelor de pe discurile magnetice este redată cu ajutorul unui arbore multicăi. Nodurile terminale reprezintă fişierele, iar nodurile neterminale – directoarele. Informaţia utilă asociată fiecărui nod include:
– numele fişierului sau directorului (`string[8]`);
– extensia (`string[3]`);
– data şi ora ultimei actualizări (respectiv ziua, luna, anul şi ore, minute, secunde);
– lungimea (`integer`);
– atributele (`'A', 'H', 'R', 'S'`).
Elaboraţi un program care simulează operaţiile de căutare, creare şi ştergere a fişierelor şi 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 depăşi acest inconvenient, tabloul respectiv poate fi înlocuit cu o listă uni- sau bidirecţională.
Elaboraţi subprogramele necesare pentru crearea şi prelucrarea arborilor multicăi de ordin arbitrar.

2.10. Tipul de date `pointer`

Acest paragraf se referă în întregime la implementarea Turbo PASCAL.

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

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

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

```
var p : pointer;
```

Întrucît astfel de declaraţii nu conţin informaţii 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 referinţă.

```
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 afişează 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 afişează 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 afişează 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 **vîrful** *heap*-ului (*fig. 2.15*).

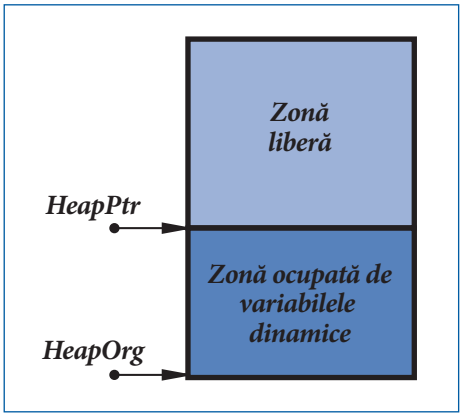

Fig. 2.15. Structura *heap*-ului

Variabilele dinamice sînt create şi 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 „încape" 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 şi vîrful listei, rădăcina arborelui etc., căutarea celorlalte componente cade în sarcina programatorului. Mai mult decît atît, 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 şi devin inaccesibile. Prin urmare, utilizarea procedurii `dispose` pentru eliberarea memoriei ocupate de structuri complexe de date este greoaie şi ineficientă. Acest inconvenient poate fi depăşit cu ajutorul procedurilor predefinite `mark` şi `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 2.16 a.* Instrucțiunea `mark(p)` a memorat în variabila de tip `pointer p` valoarea actuală din `HeapPtr` înainte de crearea variabilei dinamice `k^`.

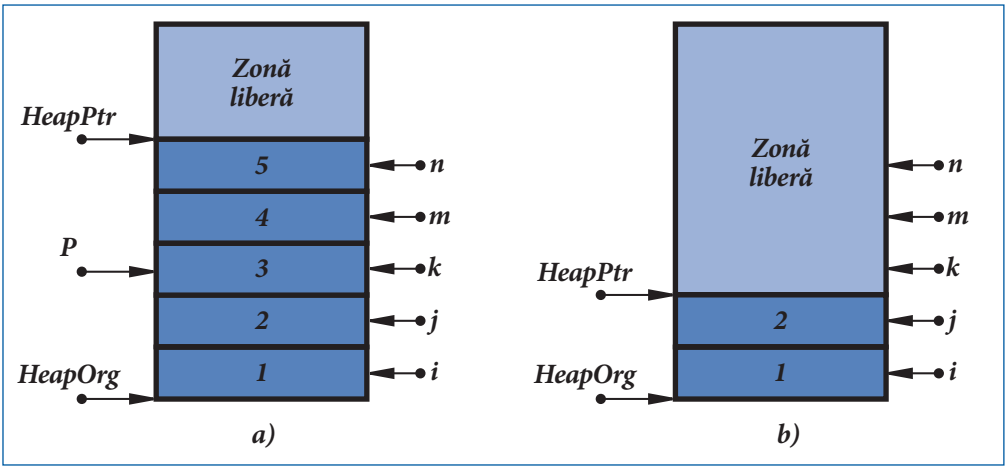

Fig. 2.16. Starea *heap*-ului pînă (a) și după executarea instrucțiunii `release(p)` (b)

Dacă acum se execută instrucțiunea

```
release(p)
```

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

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

```
release(HeapOrg)
```

Programul ce urmează ilustrează utilizarea procedurilor `mark` şi `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` şi `release` nu atribuie valoarea **`nil`** indicatorilor de adresă variabilele dinamice ale cărora au fost distruse (*fig. 2.16b*). Întrucît memoria eliberată este refolosită, atribuirile efectuate asupra variabilelor distruse pot altera valorile variabilelor dinamice nou-create.

Întrebări şi exerciţii

❶ Care este mulţimea de valori ale tipului de date `pointer`? Ce operaţii pot fi efectuate cu aceste valori?

❷ Comentaţi 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? Argumentaţi răspunsul.

❺ Lansaţi în execuţie programele ce urmează. Explicaţi rezultatele afişate 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 reţine adresa din j }
  {distrugerea variabilei j^ şi 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^ şi 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^ şi 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.
```

❻ Scrieţi o procedură care eliberează memoria ocupată de:
a) o listă unidirecţională;
b) un arbore binar;
c) un arbore multicăi.
Memoria trebuie eliberată apelînd procedura `dispose` pentru fiecare componentă a structurii dinamice de date.

❼ Scrieţi un program în care se creează mai întîi o coadă, iar apoi un arbore multicăi. Spaţiul de memorie eliberat după distrugerea cozii trebuie refolosit pentru alocarea arborelui.

Capitolul 3

METODE DE ELABORARE A PRODUSELOR PROGRAM

3.1. Programarea modulară

Programarea modulară urmăreşte reducerea complexităţii programelor mari prin descompunerea acestora în module.

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

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

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

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

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

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

Secțiunea de interfață începe cu cuvîntul-cheie **interface**. Aici se declară constantele, tipurile, variabilele şi 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 şi procedurilor exportate. Dacă unitatea actuală utilizează alte unități, numele acestora sînt specificate în clauza **uses**.

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

Secțiunea de inițializare începe, dacă există, cu cuvîntul-cheie **begin**. Secțiunea constă dintr-o secvență de instrucțiuni şi serveşte 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` şi `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 şi acelaşi 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` şi **`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*, şi 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 şi procedurilor referitoare la lucrul cu ecranul în mod *text*, precum şi cu tastatura şi difuzorul. Accesibilitatea subprogramelor se realizează prin clauza **`uses`** `crt`.

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

`Printer` — asigură redirectarea operațiilor de scriere în fişierul *text* cu numele `lst` la imprimantă. Utilizînd unitatea în studiu, programatorul nu mai trebuie să declare, să deschidă şi să închidă acest fişier. Serviciile unității `Printer` devin accesibile unui program sau unei unități de program prin specificarea clausei **`uses`** `printer`.

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

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

Întrebări şi 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 şi destinația secțiunilor de interfață, de implementare şi de inițializare.

❸ Cum se determină domeniile de vizibilitate ale declarațiilor din unitățile de program?

❹ 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.
```

❺ 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.
```

❻ Completați unitatea de program `U1` din paragraful în studiu cu un subprogram care:
a) returnează media aritmetică a componentelor unui vector;
b) aranjează componentele în ordine crescătoare;
c) returnează produsul componentelor unui vector;
d) returnează numărul componentelor pozitive;
e) aranjează componentele în ordine descrescătoare.

❼ Scrieți o unitate de program care oferă descrieri de date și subprograme pentru prelucrarea matricelor.

❽ 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:
a) citirea numerelor de la tastatură;
b) afișarea numerelor pe ecran;
c) `+`, `-`, `*`, **`mod`**, **`div`**;
d) calcularea factorialului;
e) citirea și scrierea numerelor în fișiere secvențiale.

❾ 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:
a) citirea șirurilor de la tastatură;
b) afișarea șirurilor pe ecran;
c) concatenarea șirurilor;
d) compararea lexicografică;
e) calcularea lungimii unui șir.

❿ Elaborați o unitate de program pentru prelucrarea:
a) listelor unidirecționale;
b) cozilor;
c) stivelor;

d) arborilor binari;

e) arborilor multicăi.

Utilizaţi în acest scop declaraţiile de tipuri, funcţii şi proceduri din capitolul 2.

⓫ Găsiţi în sistemul de asistenţă *Turbo PASCAL's Online Help* descrierea unităţilor-standard instalate pe calculatorul dvs. Afişaţi pe ecran textul fiecărei unităţi, determinaţi destinaţia şi modul de utilizare a funcţiilor şi procedurilor respective.

3.2. Testarea şi depanarea programelor

Un program este **corect** dacă:

a) după lansarea în execuţie procesul de calcul se termină;

b) rezultatele obţinute reprezintă o soluţie a problemei pentru rezolvarea căreia a fost scris programul.

În caz contrar programul conţine **erori**.

Asigurarea corectitudinii unui program presupune execuţia sa pentru fiecare combinaţie 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 soluţiile 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 funcţie de modul de selecţie a datelor de testare, deosebim:

– testarea funcţională 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 funcţionale** datele de testare sînt astfel concepute, încît să se asigure că fiecare funcţie a programului este pe deplin realizată. Programul este văzut ca o cutie neagră, a cărei funcţionare este determinată prin introducerea unor date şi analiza rezultatelor obţinute. Selectarea datelor de intrare depinde, în mare măsură, de îndemînarea şi experienţa celui care efectuează testarea. În mod obişnuit, se selectează **valori tipice** şi **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 funcţionale el nu este necesar. Programul realizează următoarele funcţii:

– citeşte de la tastatură un şir de numere reale;

– afişează pe ecran media aritmetică a numerelor pozitive din şir.

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

Datele de testare vor include:

a) valorile netipice:

– şir vid;

– şir ce nu conţine numere pozitive;

– şir ce conţine un singur număr pozitiv;

b) valorile tipice:
– şir cu două numere pozitive;
– şir cu trei sau mai multe numere pozitive.

În **testarea structurală** testele sînt elaborate examinînd structura programului: declarațiile de date, proceduri şi funcții, instrucțiunile simple, instrucțiunile structurate 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 şi de mai multe ori;

c) execuția fiecărei instrucțiuni **if**, **repeat**, **while** pentru valorile `true` şi `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 şir:

```
Program P143;
  { Media numerelor pozitive dintr-un şir }
var n, k : integer;
    x, s : real;
begin
  n:=0;
  k:=0;
  s:=0;
  writeln('Daţi un şir 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('şir vid')
        else if k=0 then writeln('Şirul nu conţine numere
                                  pozitive')
                    else writeln('Media=', s/k);
 readln;
end.
```

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

– un şir vid (**not eof**=`false`, `n=0`);
– un şir nevid (**not eof**=`true`, $n\neq0$);
– un şir ce conține cel puțin un număr pozitiv ($x>0$, $k\neq0$);
– un şir nevid ce nu conține numere pozitive ($x\leq0$, $k=0$).

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` şirul ce conține cel puțin un număr pozitiv poate fi înlocuit cu trei şiruri ce conțin, respectiv unul, două, trei sau mai multe numere pozitive. Aceste date vor asigura execuția instrucțiunilor `k:=k+1`, `s:=s+x` şi calcularea expresiei `s/k` pentru valorile tipice şi valorile netipice ale variabilelor `k` şi `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 combinări, în care instrucțiunile de atribuire şi apelurile de proceduri pot fi executate.

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

În metoda **depanării statice** cauzele erorii se stabilesc analizînd rezultatele derulării programului şi mesajele sistemului de operare. Pentru a facilita procesul de depanare, în program temporar se includ instrucțiuni care afişează 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 şi eliminarea unor puncte de suspendare a executării;
– modificarea valorilor unor variabile;
– trasarea apelurilor de funcții şi proceduri;
– tratarea erorilor de intrare–ieşire, a erorilor de depăşire 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 şi testat programul, calitatea mesajelor de eroare generate de calculator şi tipul erorii. De regulă, un test care semnalează prezența unei erori este urmat de alte texte organizate în aşa fel, încît să izoleze eroarea şi să furnizeze informații pentru corectarea ei.

S-a constatat că testarea şi 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 şi aplicarea metodelor programării structurate.

Subliniem faptul că testarea programelor este un mijloc eficient de a **depista erorile**, însă nu şi 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 bazate pe demonstrarea formală a corectitudinii programului, însă rezultatele cunoscute în această direcție nu sînt aplicabile programelor complexe.

Întrebări şi exerciţii

❶ Cînd un program PASCAL este corect? Cum poate fi asigurată corectitudinea unui program?

❷ Cum se selectează datele de intrare în metoda testării funcționale?

❸ Elaborați un test funcțional pentru programul `P124` din paragraful 2.4. Programul realizează următoarele funcţii:
– 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 elaborați testele funcționale:
a) `P117` şi `P120` din paragraful 2.1;
b) `P122` şi `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 Elaborați teste structurale pentru programele ce urmează:
a) `P117` şi `P120` din paragraful 2.1;
b) `P122` şi `P123` din paragraful 2.2;
c) `P127` din paragraful 2.5.

❼ Care este diferența dintre *depanarea statică* şi *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 şi modificarea programelor necesită un mare volum de muncă. Mai mult decît atît, programele complexe ce conțin sute şi mii de instrucțiuni devin greu accesibile chiar şi 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 atribuituri şi/sau apeluri de proceduri);
– decizia (**`if`**... **`then`**... sau **`if`**... **`then`**... **`else`**...);
– ciclul cu test inițial (**`while`**... **`do`**...).

Programarea structurată admite şi 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 şi unități de program trebuie să fie cît mai sugestivi.

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

6. Operațiile de intrare–ieşire 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 şi reveniri. Menționăm că, conform teoremei de structură, orice program poate fi scris fără a utiliza instrucțiunea **`goto`**. Totuşi unii autori admit utilizarea acestei instrucțiuni cu condiția ca ea să fie folosită la minimum, iar salturile să fie efectuate numai în jos.

Întrebări şi exerciții

❶ Care este justificarea teoretică a programării structurate?

❷ Precizați structurile de control necesare şi suficiente pentru reprezentarea oricărui algoritm.

❸ Formulați regulile de bază ale programării structurate.

❹ Care sînt avantajele programării structurate?

❺ Corespund oare programele `P124`, `P130` şi `P135` din capitolul 2 regulilor de bază ale programării structurate?

❻ Programul ce urmează afişează 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;
```

```
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, examinîndu-se șirurile:

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

2, 3, ..., *k*;

3, ..., *k*

ș.a.m.d., unde $k = (n + 1)$ **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.

Capitolul 4

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 şi acelaşi 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, şi în acest manual algoritmii pe care îi vom studia vor fi descrişi cu ajutorul mijloacelor oferite de limbajul PASCAL: instrucțiuni, funcții, proceduri şi programe ce pot fi derulate pe calculator.

Complexitatea algoritmului se caracterizează prin *necesarul de memorie* şi *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 şi de extragere a rezultatelor.

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

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

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

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

iar de algoritmul A_2:

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

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

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

Tabelul 4.1.

Complexitatea algoritmilor A_1 şi 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 depăşi limita impusă de mediul de programare (64 *Kocteți* pentru variabilele statice din programele Turbo PASCAL 7.0).

Determinarea necesarului de memorie $V(n)$ şi a timpului de execuție $T(n)$ prezintă un interes deosebit la etapa de elaborare a algoritmilor şi 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 şi exerciții

❶ Explicați termenul *complexitatea algoritmului*. Numiți indicatorii ce caracterizează complexitatea algoritmilor.

❷ Cum credeți, care factori influențează complexitatea unui algoritm?

❸ De ce depinde necesarul de memorie şi timpul cerut de un algoritm? Cînd este posibilă aplicarea practică a unui algoritm?

❹ Algoritmii A_1 şi 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?

❺ 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?

❻ 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 însumînd 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*
`integer`	2
`real`	6
`boolean`	1
`char`	1
enumerare	1
subdomeniu	conform tipului de bază
referință	4
`pointer`	4

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

De exemplu, necesarul de memorie pentru variabilele `A`, `B`, `p` şi `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 şi 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 şi adreselor de revenire pe durata execuției subprogramelor PASCAL. Apelul unui subprogram implică depunerea datelor respective în stivă, iar ieşirea 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 şi, eventual, distruse cu ajutorul procedurilor `new` şi `dispose`.

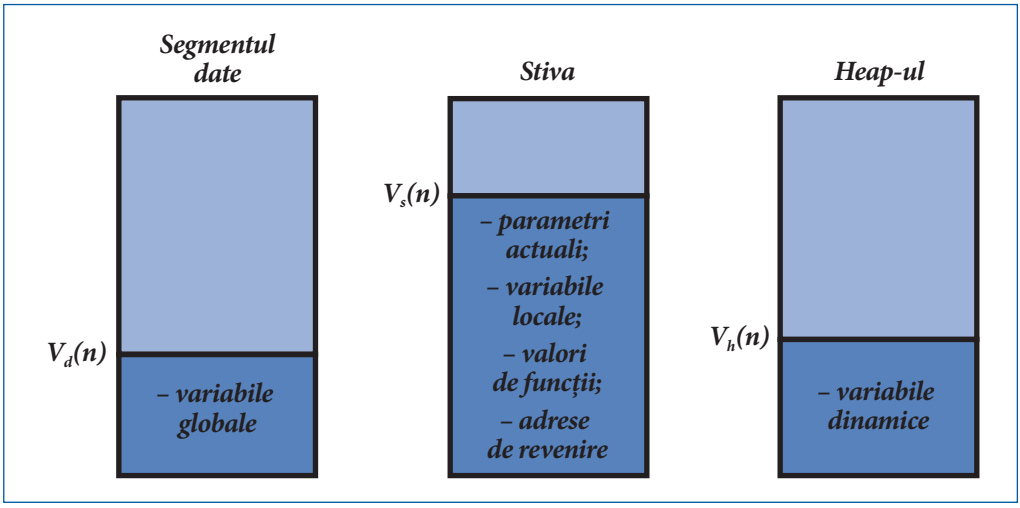

Fig. 4.1. Gestionarea memoriei interne

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 şi 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 $V_d(n) \le 64$ *Kocteți*, $V_s(n) \le 16$ *Kocteți* şi $V_h(n) \le 256$ *Kocteți*. Dimensiunile *stivei* şi 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);
  {...afişarea rezultatelor...}
 writeln('Sfîrşit');
 readln;
end.
```

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

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

Execuţia instrucţiunii apel de procedură `Pr(A)` implică depunerea în stivă a adresei matricei `A`, a adresei de revenire în programul principal şi a variabilei locale `C`. Necesarul de memorie pentru aceste date:

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

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

Instrucţiunile `new(p)` şi `new(q)` creează în *heap* variabilele dinamice `p^` şi `q^` de tipul `Matrice`. Necesarul de memorie pentru aceste variabile:

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

După execuţia instrucţiunilor `dispose(p)` şi `dispose(q)`, variabilele dinamice din *heap* sînt distruse, iar spaţiul respectiv de memorie devine liber.

Întrebări şi exerciţii

❶ Determinaţi cu ajutorul sistemului de asistenţă al mediului de programare cu care lucraţi dvs. necesarul de memorie pentru variabilele nestructurate.

❷ Cum se evaluează volumul de memorie internă necesar pentru înmagazinarea datelor unui algoritm?

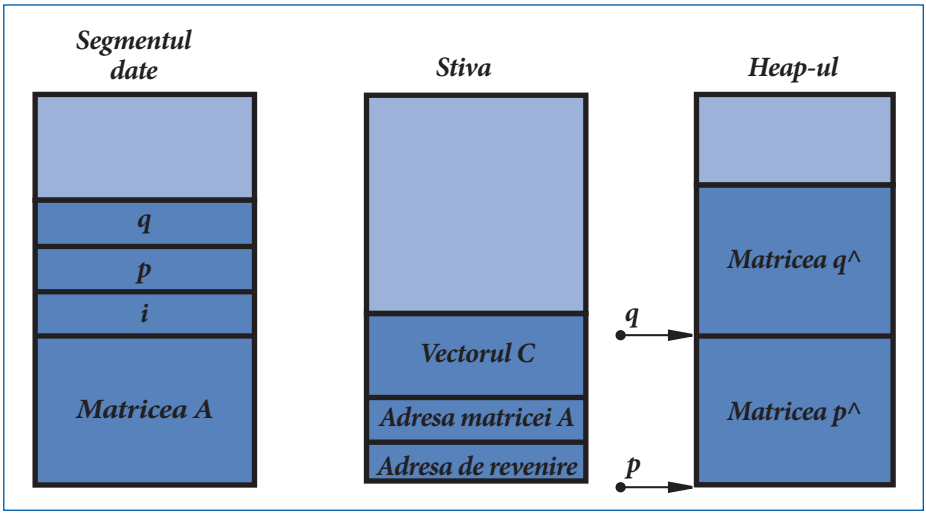

Fig. 4.2. Gestionarea memoriei în programul `P145`

❸ Explicați cum se gestionează memoria internă în cazul unui program PASCAL.

❹ 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?

❺ 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;;
```

❻ 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 şi B...}
 writeln('Sfîrşit');
 readln;
end.
```

❼ 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

$$S(n) = 0 + 1 + 2 + ... + n$$

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.

❽ 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('Sfîrşit');
 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 şi 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 efectuîndu-se n-1 comparări ale elementelor vecine `A[j]` şi `A[j+1]`. Dacă `A[j]>A[j+1]`, elementele vecine îşi schimbă locul.

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

$$A = (n, n\text{-}1, n\text{-}2, ..., 3, 2, 1).$$

De exemplu, pentru n=4, vectorul iniţial va fi

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

În procesul sortării avem:

$i = 1, \quad A = (3, 2, 1, 4);$
$i = 2, \quad A = (2, 1, 3, 4);$
$i = 3, \quad A = (1, 2, 3, 4);$
$i = 4, \quad A = (1, 2, 3, 4).$

Timpul de execuţie a procedurii `Sortare` în cazul unui calculator *Pentium* cu frecvenţa ceasului de sistem 500 *MHz* este prezentat în *tabelul 4.3*, iar graficul respectiv – în *figura 4.3*.

Tabelul 4.3

Timpul de execuţie 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 şi exerciţii

❶ Cum credeţi, ce legătură există între timpul necesar execuţiei unui program PASCAL, frecvenţa ceasului de sistem şi capacitatea de prelucrare a calculatorului?

❷ Măsuraţi timpul de execuţie a procedurii `Sortare` (vezi programul `P149`) în cazul calculatorului cu care lucraţi dvs. Construiţi un grafic similar celui din *figura 4.3*.

❸ Reprezentaţi grafic pe un singur desen timpul de execuţie 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;
```

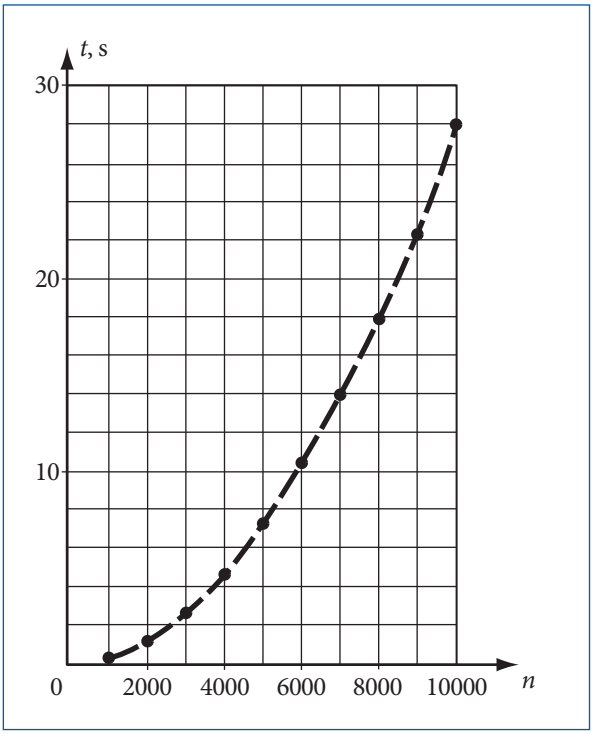

Fig. 4.3. Timpul de execuție a procedurii `Sortare`

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

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; { N4 }
```

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

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

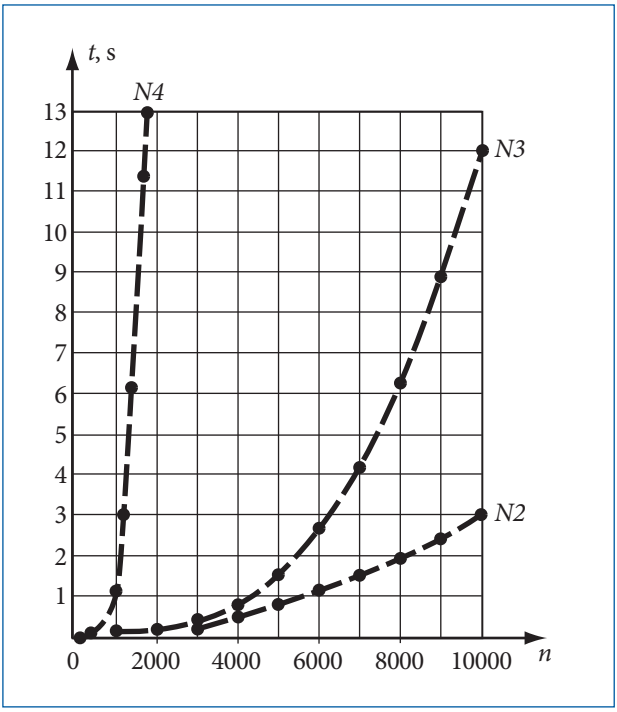

Fig. 4.4. Timpul de execuție a procedurilor `N2`, `N3` și `N4`

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 şi 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. Acelaşi timp Δ este necesar pentru indexarea componentelor unui tablou [ ], pentru atribuirea **:=** şi instrucțiunea **goto**. Valoarea concretă a mărimii Δ depinde de mediul de programare, capacitatea de prelucrare a calculatorului utilizat şi este de ordinul 10^{-9} ... 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 şi 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:

	Expresia E	*Numărul de operații elementare* Q_E
a)	`a*b+c`	2
b)	`(a<b) 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_1; I_2; ...; I_k **end**	$Q_E + \max\{Q_{I_1}, Q_{I_2}, ..., Q_{I_k}\} + k + 1$
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; ...; I_k **end**	$Q_{I_1} + Q_{I_2} + ... + Q_{I_k} + 1$
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 şi cu ajutorul instrucțiunilor **if** şi **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, ... , I_8 ale procedurii `Sortare` vor fi referite cu ajutorul comentariilor {1}, {2}, ..., {8} din partea stîngă a liniilor de program. Prin Q_j vom nota numărul de operații elementare necesare pentru executarea instrucțiunii I_j:

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

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 întîi se analizează instrucțiunile simple, iar apoi cele structurate. În cazul instrucțiunilor imbricate, mai întîi 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*) $n = 10000$ şi $T(n) = 28{,}18$ *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 şi 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 şi 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_i 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; { 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; { N4 }
```

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

❺ În procesul compilării, instrucțiunile limbajului PASCAL sînt translatate 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 translatată fiecare instrucțiune PASCAL.

❻ 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.

❼ 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 $Q(n)$. Întrucît calculatoarele moderne au o viteză de calcul foarte mare – 10^8 ... 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 cît 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, algoritmii se clasifică în:
– algoritmi polinomiali;
– algoritmi exponențiali;
– algoritmi nederminist polinomiali.

Un algoritm se numeşte **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 citeşte „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 şi din graficul timpului de execuţie $T(n)$ a procedurii `Sortare`, grafic prezentat în *figura 4.3*.

Un algoritm se numeşte **exponenţial** 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 exponenţiali este redată prin notaţia $O(k^n)$.

Menţionăm faptul că tot exponenţiali se consideră şi algoritmii de complexitatea $n^{\log n}$, cu toate că această funcţie nu este exponenţială în sensul strict matematic al acestui cuvînt.

Din comparaţia vitezei de creştere a funcţiilor exponenţială şi polinomială (vezi *tabelul 4.5*) rezultă că algoritmii exponenţiali devin inutilizabili chiar pentru valori nu prea mari ale lui n. Pentru exemplificare amintim *tabelul 4.1* în care este prezentat timpul de execuţie $T_1(n)$ a unui algoritm polinomial de ordinul $O(n^3)$ şi timpul de execuţie $T_2(n)$ a unui algoritm exponenţial de ordinul $O(2^n)$.

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

În funcţie de complexitatea temporală se consideră că o problemă este **uşor rezolvabilă** dacă pentru soluţionarea ei există un algoritm polinomial. O problemă pentru care nu există un algoritm polinomial se numeşte **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 situaţia poate fi diferită.

De exemplu, presupunem că pentru soluţionarea unei probleme există doi algoritmi, unul polinomial cu timpul de execuţie $T_1(n)$ şi altul exponenţial cu timpul de execuţie $T_2(n)$:

$$T_1(n) = 1000n^2\Delta;$$
$$T_2(n) = 2^n\Delta.$$

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

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

– formularea exactă a problemei;

– determinarea complexităţii temporale a problemei propuse – problemă uşor rezolvabilă sau problemă dificilă;

– elaborarea algoritmului respectiv şi implementarea lui pe un sistem de calcul.

Evident, în cazul unor probleme uşor rezolvabile, programatorul va depune toate eforturile pentru a inventa algoritmi polinomiali, adică algoritmi de ordinul n^k, astfel încît parametrul k să ia valori cît mai mici. În cazul problemelor dificile se va da prioritate algoritmilor care minimizează timpul de execuţie cel puţin pentru datele de intrare frecvent utilizate în aplicaţiile practice. Metodele de clasificare a problemelor în cele uşor şi cele dificil rezolvabile se studiază în cursurile avansate de informatică.

Întrebări şi exerciţii

❶ Indicaţi termenii dominanţi:

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 funcţie de complexitatea temporală?

❸ Determinaţi tipul algoritmilor, cunoscînd 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 credeţi, prin ce se explică importanţa termenului dominant în analiza complexităţii temporale a algoritmilor?

❺ Se consideră procedurile `N2`, `N3` şi `N4` din paragraful precedent. În urma estimării numărului de operaţii elementare s-a obţinut:

$$Q_{N2}(n) = 602n^2 + 2n + 2;$$
$$Q_{N3}(n) = 2n^3 + 2n^2 + 2n + 2;$$
$$Q_{N4}(n) = 2n^4 + 2n^3 + 2n^2 + 2n + 2.$$

Determinaţi ordinul de complexitate temporală a algoritmilor descrişi cu ajutorul acestor proceduri. Comparaţi viteza de creştere a timpilor de execuţie $T_{N2}(n)$, $T_{N3}(n)$ şi $T_{N4}(n)$, folosind în acest scop graficele din *figura 4.4*.

❻ Se consideră un algoritm format din k cicluri imbricate:

```
for i1:=1 to n do
   for i2:=1 to n do
       ...
           for ik:=1 to n do P
```

Numărul de operații elementare Q_p efectuate în procedura `P` este o mărime constantă. Estimați complexitatea temporală a algoritmului.

❼ 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.

Capitolul 5

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 ş.a.

Una din cele mai răspîndite tehnici de programare este **recursia**. Amintim că recursia se defineşte ca o situație în care un subprogram se autoapelează fie direct, fie prin intermediul altui subprogram. Tehnicile în studiu se numesc respectiv **recursia directă** şi **recursia indirectă** şi au fost studiate în cadrul temei „Funcții şi proceduri".

În general, elaborarea unui program recursiv este posibilă numai atunci cînd 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 calculelor trebuie să existe:

– cazuri elementare, care se rezolvă direct;

– cazuri care nu se rezolvă direct, însă procesul de calcul în mod obligatoriu progresează spre un caz elementar.

De exemplu, în definiția recursivă a funcției factorial *fact*: $\boldsymbol{N}\rightarrow\boldsymbol{N}$,

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

deosebim:

− Cazul elementar $n = 0$. În acest caz valoarea $fact(0)$ este direct calculabilă şi anume $fact(0)=1$.

− Cazurile neelementare $n > 0$. În astfel de cazuri valorile $fact(n)$ nu sînt direct calculabile, însă procesul de calcul progresează 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*.

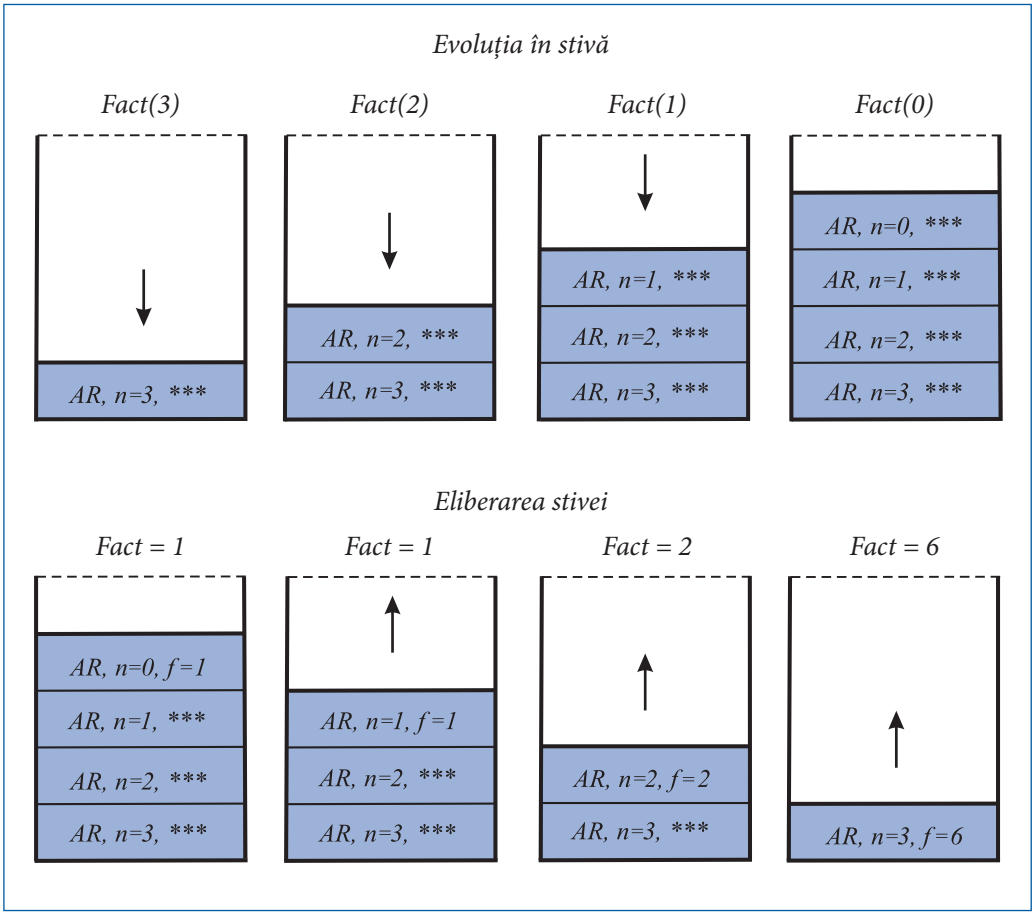

Fig. 5.1. Gestionarea stivei în cazul apelului `Fact(3)`:
AR – adresa de revenire; *n* – valoarea curentă a parametrului actual;
*** – spațiu pentru memorarea valorii *f* returnate de funcția `Fact`

Într-un mod similar, în definiția recursivă a funcției *incons*: $N \to N$,

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

deosebim cazul elementar $n=0$ şi cazurile neelementare $n>0$. Însă, spre deosebire de funcția *fact*(n), pentru $n > 0$ valorile *incons*(n) nu pot fi calculate, întrucît procesul de calcul progresează către *incons*(∞).

De exemplu, pentru n=3 obținem:

$$incons(3) = 3 \cdot incons(4) = 3 \cdot 4 \cdot incons(5) = 3 \cdot 4 \cdot 5 \cdot incons(6) = \cdots.$$

Prin urmare, definiția recursivă a funcției *incons*(n) este o **definiție inconsistentă** şi teoretic procesul de calcul va dura la nesfîrşit. În practică calculele vor fi întrerupte de sistemul de operare în momentul depăşirii capacității de memorare a stivei sau în cazul depăşirii 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 şi invers. Alegerea tehnicii de programare – **iterativitate** sau **recursivitate** – ține, de asemenea, de competența programatorului. Evident, această alegere trebuie făcută luînd în considerare avantajele şi 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 şi recursivității (prelucrarea automată a textelor)

Nr. crt.	*Caracteristici*	*Iterativitate*	*Recursivitate*
1.	Necesarul de memorie	mic	mare
2.	Timpul de execuție	acelaşi	
3.	Structura programului	complicată	simplă
4.	Volumul de muncă necesar pentru scrierea programului	mare	mic
5.	Testarea şi depanarea programelor	simplă	complicată

Propunem cititorului în calitate de exercițiu efectuarea unui studiu comparativ al iterativității şi recursivității în cazul algoritmilor destinați creării şi 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 ş.a. Un exemplu tipic de astfel de probleme este analiza gramaticală a programelor PASCAL, sintaxa cărora, după cum se ştie, este definită prin relații de recurență.

Întrebări şi exerciții

❶ Explicați termenul *tehnici de programare*.

❷ Care este diferența dintre *recursia directă* şi *recursia indirectă*? Dați exemple.

❸ Ce condiții trebuie respectate pentru ca definiția unui algoritm recursiv să fie corectă?

❹ Care este diferența dintre definițiile recursive consistente şi definițiile recursive inconsistente?

❺ Se consideră următoarele definiții recursive de funcții. Care din aceste definiții sînt consistente? Argumentați răspunsul.

a) $f: \mathbf{N} \to \mathbf{N}$, $f(n) = \begin{cases} 1, \text{ dacă } n = 0; \\ n + f(n-1), \text{ dacă } n > 0; \end{cases}$

b) $f: \mathbf{N} \to \mathbf{N}$, $f(n) = \begin{cases} 1, \text{ dacă } n = 0; \\ n + f(n), \text{ dacă } n > 0; \end{cases}$

c) $f: \mathbf{Z} \to \mathbf{Z}$, $f(i) = \begin{cases} 1, \text{ dacă } i = 0; \\ i + f(i-1), \text{ dacă } i \neq 0; \end{cases}$

d) $f: \mathbf{N} \to \mathbf{N}$, $f(n) = \begin{cases} 1, \text{ dacă } n = 0; \\ n + g(n-1), \text{ dacă } n > 0; \end{cases}$

$g: \mathbf{N} \to \mathbf{N}$, $g(n) = \begin{cases} 2, \text{ dacă } n = 0; \\ n + f(n-1), \text{ dacă } n > 0; \end{cases}$

e) $f: \mathbf{N} \to \mathbf{N}$, $f(n) = \begin{cases} n, \text{ dacă } n < 10; \\ (n \bmod 10) + f(n \operatorname{div} 10), \text{ dacă } n \geq 10. \end{cases}$

❻ Lansați în execuție programul ce urmează. Explicați mesajele afișate la ecran.

```
Program P150;
 { Depăşirea capacităţii de memorare a stivei }
type Natural = 0..Maxint;

function Incons(n:Natural):Natural;
 { Definiţie 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).

❼ Indicați cazurile elementare și cele neelementare pentru următorii algoritmi recursivi:

a) funcția `Arb` și procedura `AfisArb` din programul `P130`;

b) procedurile `Preordine`, `Inordine` și `Postordine` din programul `P131`;

c) procedura `InAdincime` din programul `P133`.

❽ Suma primelor *n* numere naturale *S*(*n*) poate fi calculată cu ajutorul funcției iterative

$$S(n) = 0 + 1 + 2 + \ldots + n = \sum_{i=o}^{n} i$$

sau al funcției recursive

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

Utilizînd ca model *tabelul 5.1,* efectuați un studiu comparativ al algoritmilor destinați calculării sumei *S*(*n*).

❾ Se consideră următoarele formule metalingvistice:
<*Cifră*> ::= `0|1|2|3|4|5|6|7|8|9`
<*Număr*> ::= <*Cifră* > {<*Cifră*>}
<*Semn*> ::= `+|-|*|/`
<*Expresie*> ::= <*Număr*> | `(`<*Expresie*>`)` |<*Expresie*><*Semn*><*Expresie*>
Scrieți o funcție recursivă care returnează valoarea `true` dacă șirul de caractere *S* este conform definiției unității lexicale <*Expresie*> și `false` în caz contrar.

❿ Efectuați un studiu comparativ al algoritmilor iterativi și algoritmilor recursivi destinați creări și prelucrării următoarelor structuri dinamice de date:
a) liste unidirecționale;
b) stiva;
c) cozi;
d) arbori binari;
e) arbori de ordinul *m*.

⓫ Scrieți o funcție iterativă care returnează valoarea `true` dacă șirul de caractere *S* este conform definiției unității lexicale <*Expresie*> din *exercițiul 9* și `false` în caz contrar. Utilizînd ca model *tabelul 5.1*, efectuați un studiu comparativ al algoritmului iterativ și algoritmului recursiv.

⓬ Elaborați un subprogram recursiv care calculează suma cifrelor unui număr natural *n*. Examinați cazurile $n \le$ `MaxInt` și $n \le 10^{250}$.

⓭ Imaginile în alb-negru (*fig. 5.2*) pot fi codificate cu ajutorul unei matrice binare B = $= \|b_{ij}\|_{n\times m}$, $1 \le n, m \le 30$. Elementul b_{ij} indică culoarea microzonei respective: neagră (b_{ij} =1) sau albă (b_{ij} =0).

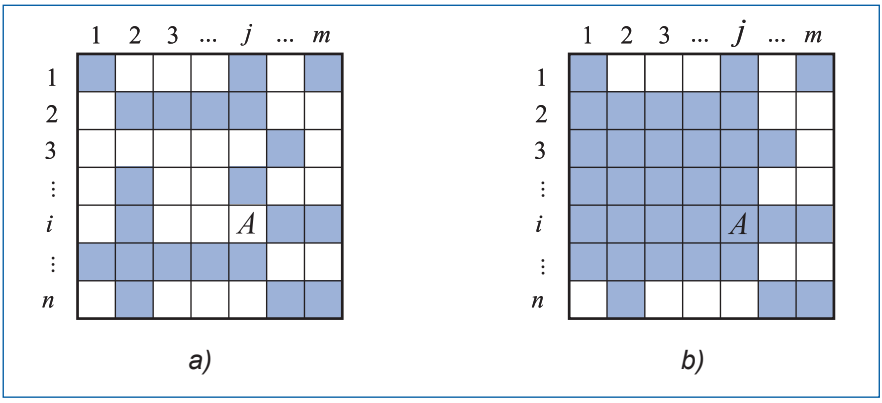

Fig. 5.2. Colorarea unei suprafețe închise:
a – imaginea inițială; *b* – imaginea finală

Elaborați o procedură recursivă pentru colorarea unei suprafețe închise, specificate prin coordonatele (*i, j*) ale oricărei microzone *A* din interiorul ei.

⓮ Elaborați o procedură care determină cîte 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).

⓯ 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 analizînd consecutiv elementele s_i ale unei mulțimi finite

$$S = \{s_1, s_2, \ldots, s_i, \ldots, s_k\},$$

denumită **mulțimea soluțiilor posibile**. În cele mai simple cazuri elementele s_i, $s_i \in S$, pot fi reprezentate prin valori aparținînd 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 $s_1, s_2, \ldots, s_k$ nu sînt indicate explicit în enunțul problemei şi 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 şi `false` în caz contrar, iar `PrelucrareaSolutiei` este o procedură care efectuează prelucrarea elementului selectat. De obicei, în această procedură soluția s_i este afişată la ecran.

În continuare vom analiza două exemple care pun în evidență avantajele şi neajunsurile algoritmilor bazați pe metoda trierii.

Exemplul 1. Se consideră numerele naturale din mulțimea {0, 1, 2, ..., n}. Elaborați un program care determină pentru cîte numere K din această mulțime suma cifrelor fiecărui număr este egală cu m. În particular, pentru n = 100 şi m = 2, în mulțimea {0, 1, 2, …, 100} există 3 numere care satisfac condițiile problemei: 2, 11 şi 20. Prin urmare, K = 3.

Rezolvare. Evident, mulțimea soluțiilor posibile S = {0, 1, 2, …, 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 şi 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 $P = \{P_1, P_2, \ldots, P_n\}$ formată din n puncte ($2 \le n \le 30$) 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(Pj, Pm) then PrelucrareaSolutiei(Pj, Pm)
```

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 algoritmii 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, aceşti algoritmi realizează operațiile legate de prelucrarea unor mulțimi:

– reuniunea;
– intersecția;
– diferența;
– generarea tuturor submulțimilor;
– generarea elementelor unui produs cartezian;
– generarea permutărilor, aranjamentelor sau combinărilor 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 algoritmii exponențiali. Întrucît algoritmii 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 şi 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 micşoreze 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 $P = \{P_1, P_2, \ldots, P_n\}$ formată din n puncte ($3 \le n \le 30$) 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.

❽ 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 necunoscuta. Fiecare cifră a numărului natural a aparține mulțimii {0, 1, 2, …, 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.

❾ 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.

❿ Elaborați un program care determină cîte puncte cu coordonate întregi se conțin într-o sferă de raza R cu centrul în originea sistemului de coordonate. Se consideră că R este un număr natural, $1 \le R \le 30$. 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 $A=\{a_1, a_2, ..., a_n\}$ 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 algoritmii 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 şi 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 şi 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 şi denumirea metodei (*greedy* – lacom, hrăpăreț).

Exemplu. Se consideră mulțimea $A=\{a_1, a_2, ..., a_i, ..., a_n\}$ 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 şi B sînt reprezentate prin vectorii (tablourile unidimensionale) `A` şi `B`, iar numărul de elemente ale fiecărei mulțimi – prin variabilele întregi, respectiv `n` şi `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` şi `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 şi exerciții

❶ Explicați structura generală a algoritmilor bazați pe metoda *Greedy*.

❷ Care sînt avantajele şi neajunsurile algoritmilor bazați pe tehnica *Greedy*?

❸ Estimați timpul de execuție al programului `P153`.

❹ Schițați un algoritm care determină submulțimea B din exemplul de mai sus prin metoda trierii. Estimați complexitatea temporală a algoritmului elaborat.

❺ **Memorarea fișierelor pe benzi magnetice.** Se consideră n fișiere $f_1, f_2, ..., f_n$ care trebuie memorate pe o bandă magnetică. Elaborați un program care determină ordinea de amplasare a fișierelor pe bandă astfel încît 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, ..., n) sînt necesare t_i secunde.

❻ **Problema continuă a rucsacului.** Se consideră n obiecte. Pentru fiecare obiect i (i=1, 2, ..., 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.

❼ **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.

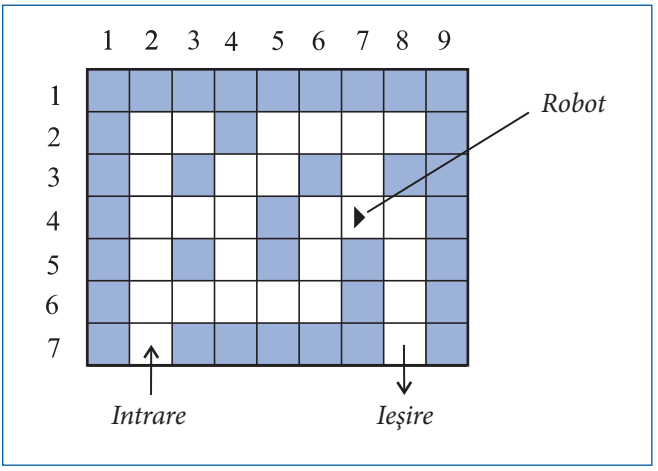

Fig. 5.3. Cîmpul de acțiune al robotului

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 haşurate reprezintă obstacolele, iar cele nehaşurate – spațiile libere. Pătrățelele de pe perimetrul planului, cu excepția celor de intrare sau ieșire, sînt haşurate 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 serveşte 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 – ieşirea din hrube. Iniţial, robotul se află în pătrăţelul pentru care $A[i, j]=2$.
Fişierul `HRUBE.IN` conţine pe prima linie numerele m, n separate prin spaţiu. Pe următoarele m linii se conţin cîte n numere $A[i, j]$ separate prin spaţiu.
Datele de ieşire. Fişierul `HRUBE.OUT` va conţine pe fiecare linie cîte una din instrucţiunile `SUS`, `JOS`, `DREAPTA`, `STINGA` scrise în ordinea executării lor de către robot.
Restricţii. $5 \le m, n \le 100$. Timpul de execuţie nu va depăşi 3 secunde.
Exemplu:

`HRUBE.IN`

```
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
```

`HRUBE.OUT`

```
SUS
DREAPTA
DREAPTA
DREAPTA
DREAPTA
SUS
SUS
DREAPTA
DREAPTA
JOS
JOS
JOS
```

Indicaţii. La fiecare pas selectaţi din mulţimea {`SUS`, `JOS`, `DREAPTA`, `STINGA`} cîte o instrucţiune în aşa fel încît robotul să se deplaseze numai de-a lungul unui perete.

5.4. Metoda reluării

În metoda reluării se presupune că soluţia problemei pe care trebuie să o rezolvăm poate fi reprezentată printr-un vector:

$$X=(x_1, x_2, ..., x_k, ... , x_n).$$

Fiecare componentă x_k a vectorului X poate lua valori dintr-o anumită mulţime A_k, $k = 1, 2, ..., n$. Se consideră că cele m_k elemente ale fiecărei mulţimi A_k sînt **ordonate** conform unui criteriu bine stabilit, de exemplu, în ordinea apariţiei lor în memoria calculatorului.

În principiu, astfel de probleme pot fi soluţionate prin metoda trierii, tratînd vectorul X ca un element al produsului cartezian $S = A_1 \times A_2 \times ... \times A_n$. Dezavantajul metodei trierii constă în faptul că timpul cerut de algoritmul respectiv este foarte mare. De exemplu, pentru mulţimile $A_1, A_2, ..., A_n$ formate numai din cîte două elemente, timpul necesar este $O(2^n)$, deci exponenţial.

Pentru evitarea trierii tuturor elementelor produsului cartezian $A_1 \times A_2 \times ... \times A_n$, 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 $x_1, x_2, ..., x_{k-1}$. 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 **condiţii de continuare** referitoare la $x_1, x_2, ..., x_k$. 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 micşorăm pe k cu o unitate încercînd să facem o nouă alegere pentru x_{k-1}.

Menționăm faptul că anume micşorarea lui k dă nume metodei studiate, cuvîntul „reluare" semnificînd revenirea la alte variante de alegere a variabilelor $x_1, x_2, ..., x_{k-1}$. Evident, aceeaşi semnificație o are şi denumirea engleză a metodei în studiu – *backtracking* (*back* – înapoi, *track* – urmă).

Pentru exemplificare, în *figura 5.4* este prezentată ordinea în care sînt examinate elementele mulțimilor $A_1 = \{a_{11}, a_{12}\}$, $A_2 = \{a_{21}, a_{22}\}$ şi $A_3 = \{a_{31}, a_{32}, a_{33}\}$. În scopuri didactice se consideră că mulțimile A_1 şi A_2 au cîte 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 cerculețe. Rezultatele verificării condițiilor de continuare sînt reprezentate prin valorile binare 0 (`false`) şi 1 (`true`).

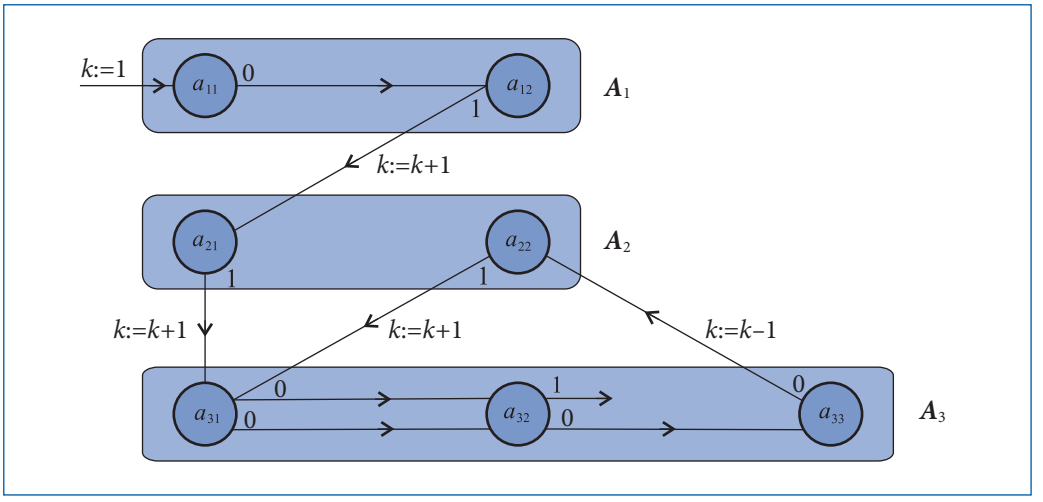

Fig. 5.4. Căutarea soluției prin metoda reluării

Din *figura 5.4* se observă că primul element a_{11} din mulțimea A_1 nu satisface condițiile de continuare şi, în consecință, se trece la elementul al doilea a_{12} din aceeaşi mulțime. Mai departe în vectorul X se include primul element a_{21} din mulțimea A_2, element care satisface condițiile de continuare, şi 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, şi 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 şi subprogramele apelate prin variabilele globale ce reprezintă vectorul X şi mulţimile A_1, A_2, ..., A_n. Subprogramele apelate execută următoarele operaţii:

`PrimulElement(k)` – returnează primul element din mulţimea A_k;

`Continuare(k)` – returnează valoarea `true` dacă elementele înscrise în primele k componente ale vectorului X satisfac condiţiile de continuare şi `false` în caz contrar;

`ExistaSuccesor(k)` – returnează valoarea `true` dacă elementul memorat în componenta x_k are un succesor în mulţimea A_k şi `false` în caz contrar;

`Succesor(k)` – returnează succesorul elementului memorat în componenta x_k;

`PrelucrareaSolutiei` – de obicei, în această procedură soluţia reţinută în vectorul X este afişată la ecran.

Pentru o înţelegere mai clară a procesului recursiv, vom urmări execuţia procedurii `Reluare` în cazul mulţimilor A_1, A_2, şi A_3 din *figura 5.3*.

La execuţia apelului `Reluare(1)` în componenta x_1 a vectorului X se înscrie primul element al mulţimii A_1:

$$X=(a_{11}).$$

Întrucît în cazul acestui vector apelul `Continuare(1)` returnează valoarea `false`, se trece la execuţia instrucţiunii **`while`**. Apelul `ExistaSuccesor(1)` returnează valoarea `true` şi, 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` şi, prin urmare, se trece la execuţia recursivă a apelului `Reluare(2)`. Evident, în componenta x_2 a vectorului X va fi înscris primul element din mulţimea A_2:

$$X=(a_{12}, a_{21}).$$

Pentru acest vector funcţia `Continuare` returnează valoarea `true`, fapt ce declanşează execuţia recursivă a apelului `Reluare(3)`.

În cazul apelului `Reluare(3)` componenta x_3 a vectorului X ia recursiv valorile a_{31}, a_{32} şi 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 aceşti vectori nu satisface condiţiile de continuare. După examinarea ultimului element din mulţimea A_3, funcţia `ExistaSuccesor` returnează valoarea `false` şi, în consecinţă, se revine în contextul apelului `Reluare(2)`. În acest context în componenta x_2 a vectorului X se înscrie succesorul elementului a_{21}:

$$X=(a_{12}, a_{22}).$$

Întrucît şi pentru acest vector funcţia `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 condiţiile de continuare, fapt ce declanşează execuţia apelului recursiv `Reluare(4)`. În acest apel $k>n$ şi, prin urmare, procedura `PrelucrareaSolutiei` va afişa coordonatele vectorului X la ecran.

Exemplu. Se consideră mulţimile $A_1, A_2, ..., A_n$, fiecare mulţime fiind formată din m_k numere naturale. Selectaţi din fiecare mulţime cîte un număr în aşa mod încît suma lor să fie egală cu q.

Rezolvare. Vom reprezenta mulţimile $A_1, A_2, ..., A_n$ prin liniile tabloului bidimensional (matricei) $A = ||a_{kj}||$. Numărul de elemente m_k al fiecărei mulţimi A_k va fi reţinut în componenta respectivă a tabloului unidimensional (vectorului) $M = ||m_k||$.

Din enunţul problemei rezultă că toate condiţiile de continuare sînt respectate numai atunci cînd suma elementelor referite de primele k componente ale vectorului X nu depăşeşte valoarea q pentru $k<n$ şi 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 mulţimile $A_1, A_2, ..., A_n$.

```
Program P154;
 { Program bazat pe metoda reluării }
const mmax=50; { numărul maximal de mulţimi }
      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 mulţimi }
     M : array[1..nmax] of 1..mmax; { cardinalul mulţimii 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 reluarii - 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` şi a ordinii de parcurgere a elementelor din mulțimile A_1, A_2, ..., A_n (*fig. 5.4*) rezultă că în cazul cel mai nefavorabil vor fi generați toți vectorii X ai produsului cartezian $A_1 \times A_2 \times ... \times A_n$. Prin urmare, **complexitatea temporală** a algoritmilor bazați pe metoda reluării este, ca şi în cazul trierii, $O(m^n)$, unde $m=\max(m_1, m_2, ..., m_n)$.

Timpul real de execuție, cerut de un algoritm, depinde în mare măsură de natura problemei, mai exact, de esența condițiilor de continuare şi ordinea în care apar elementele a_{kj} în mulțimile A_1, A_2, ..., 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, ..., A_n va fi testat numai cîte un singur element şi, 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 întîlnită în practică. Identificarea acestor condiții şi, în consecință, reducerea timpului de calcul depinde de iscusința programatorului. Menționăm că în majoritatea problemelor mulțimile A_1, A_2, ..., A_n nu sînt cunoscute apriori, iar componentele vectorului X pot aparține şi ele unor tipuri structurate de date – tablouri, mulțimi sau articole.

Întrebări şi exerciții

❶ Explicați structura generală a algoritmilor bazați pe metoda reluării.

❷ Indicați cazurile elementare şi cele neelementare în procedura recursivă `Reluare`.

❸ Elaborați o variantă iterativă a procedurii `Reluare`.

❹ Elaborați un studiu comparativ al algoritmilor iterativi şi 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, ..., 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) $A_1=\{1, 2\}$, $A_2=\{3, 4\}$, $A_3=\{5, 6\}$, $A_4=\{7, 8\}$, $q=20$.

❻ Precizați condițiile de continuare pentru programele în care se dorește generarea tuturor elementelor produsului cartezian $A_1 \times A_2 \times ... \times A_n$. Pentru a verifica răspunsul scrieți și depanați programul respectiv.

❼ 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.

❽ Se consideră mulțimea $B=\{b_1, b_2, ..., b_n\}$ 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 $x_1+x_2+...+x_n=q$.

❾ **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.

❿ **Labirintul.** Se consideră planul unui labirint desenat pe o foaie de hîrtie liniată în pătrățele (*fig. 5.5*). Pătrățelele haşurate reprezintă obstacolele, iar cele nehaşurate – camerele și coridoarele labirintului.

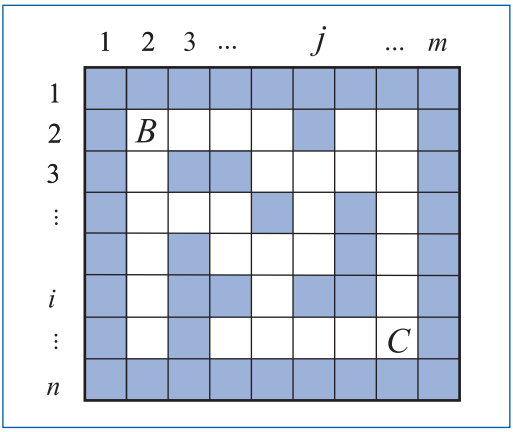

Fig. 5.5. Planul unui labirint

În memoria calculatorului planul labirintului este redat prin matricea $A=||a_{ij}||_{n \times m}$, $1 \leq n,m \leq 30$, unde

$$a_{ij} = \begin{cases} 1, & \text{dacă pătrățelul } (i, j) \text{ este haşurat;} \\ 0, & \text{în caz contrar.} \end{cases}$$

Călătorul se poate deplasa dintr-un pătrățel nehaşurat în alt pătrățel nehaşurat numai atunci cînd ele au o latură comună. Elaborați un program, care găsește, dacă există, un drum din pătrățelul *B* în pătrățelul *C*.

⓫ Se consideră n ($n \leq 30$) săculețe, numerotate prin 1, 2, 3, ..., 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.

⓬ 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.

⓭ 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* şi *stăpîneşte*

Metoda *desparte* şi *stăpîneşte* (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 şi 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

$$A = (a_i, a_{i+1}, ..., a_j)$$

şi că trebuie efectuată o prelucrare oarecare asupra elementelor sale. Pentru a împărți problema curentă în două subprobleme de aproximativ aceleaşi dimensiuni, stabilim

$$m = (j - i) \text{ div } 2$$

şi împărțim mulțimea A în două submulțimi care vor fi prelucrate separat:

$$A_1 = (a_i, a_{i+1}, ..., a_{i+m}); \; A_2 = (a_{i+m+1}, a_{i+m+2}, ..., a_j).$$

În continuare, mulțimile A_1 şi A_2 se împart din nou în cîte două submulțimi, respectiv $A_{1\text{-}1}$, $A_{1\text{-}2}$ şi $A_{2\text{-}1}$, $A_{2\text{-}2}$. 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 $A=(a_1, a_2, ..., a_7)$ în cazul divizării problemelor curente în cîte două subprobleme de același tip.

Schema generală a unui algoritm bazat pe metoda *desparte* şi *stăpîneşte* 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;
```

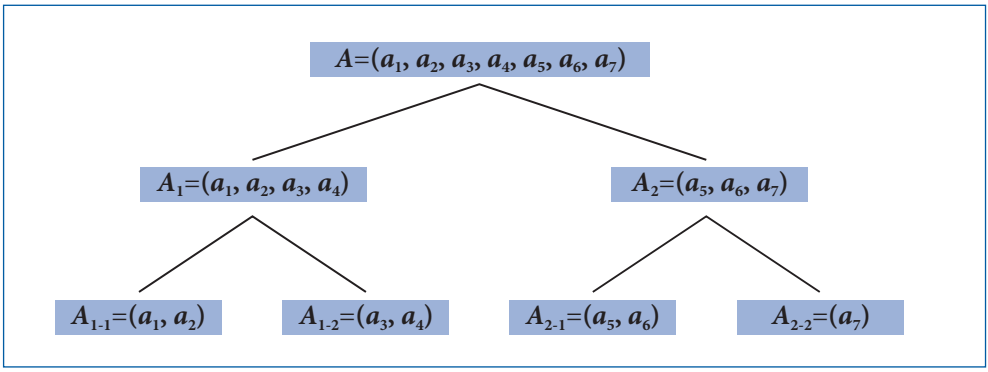

Fig. 5.6. Descompunerea mulțimii *A* în submulțimi

Parametrii `i` şi `j` din antetul procedurii definesc submulțimea curentă (a_i, a_{i+1}, ..., a_j) supusă prelucrării. Funcția `SolutieDirecta` returnează valoarea `true` dacă subproblema curentă admite o rezolvare directă şi `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}, ..., a_m) şi altul pentru submulțimea (a_{i+m+1}, a_{i+m+2}, ..., a_j). Procedura `Combina` prelucrează soluțiile `x1` şi `x2` ale subproblemelor respective şi returnează soluția `x` a problemei curente.

Exemplul 1. Se consideră mulțimea $A=\{a_1, a_2, ..., a_n\}$ 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 cînd mulțimea (a_i, ..., 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 elementuli maximal prin metoda desparte şi
stăpîneşte }
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* şi *stăpîneşte* admite împărțirea problemelor curente într-un număr arbitrar de subprobleme, nu neapărat în două. În

exemplul ce urmează problema curentă – tăierea unei placi de arie maximă – este împărțită în patru subprobleme de acelaşi 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). Elaborați un program care decupează din placă o bucată de arie maximă, dreptunghiulară şi fără găuri. Sînt admise doar tăieturi de la o margine la alta pe direcții paralele cu laturile plăcii – verticale sau orizontale (*fig. 5.7*).

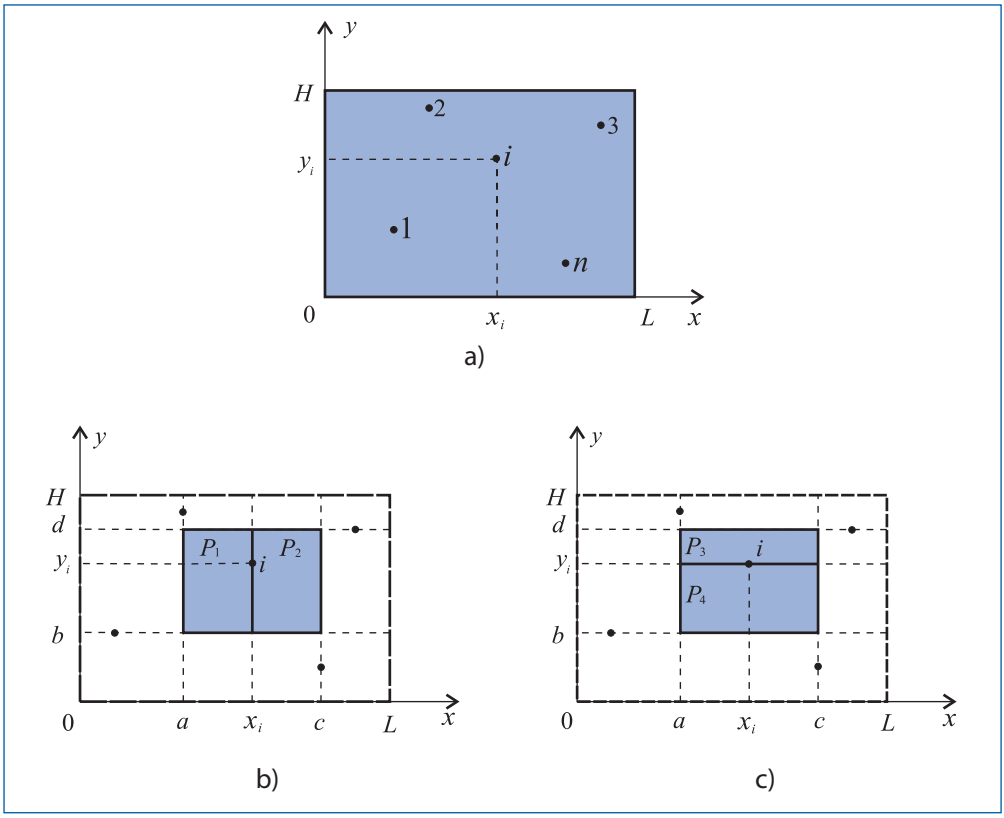

Fig. 5.7. Tăierea unei plăci de arie maximă:
a – placa inițială; *b* – tăierea pe verticală; *c* – tăierea pe orizontală

Rezolvare. Vom defini placa curentă prin vectorul $P=(a, b, c, d)$, unde a şi b sînt coordonatele colțului stînga-jos, iar c şi d – coordonatele colțului dreapta-sus. Evident, placa inițială se defineşte prin $(0, 0, L, H)$. Metoda *desparte* şi *stăpîneşte* 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*:

$$P_1=(a, b, x_i, d),\ P_2=(x_i, b, c, d) \quad \text{sau} \quad P_3=(a, y_i, c, d),\ P_4=(a, b, c, y_i);$$

– în continuare examinăm în acelaşi mod fiecare din plăcile obţinute în urma tăierii, memorînd consecutiv în variabila S_{max} aria plăcii de suprafaţă maximă.

```
Program P156;
 { Tăierea unei placi prin metoda desparte şi stăpîneşte }
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 1;
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 1;
      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;
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 şi `false` în caz contrar. În cazul valorii `false`, funcția returnează suplimentar numărul de ordine i al uneia din găurile plăcii. Această gaură este depistată verificînd relațiile $a<x_i<c$ şi $b<y_i<$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* şi *stăpîneşte* depinde de numărul de subprobleme k în care este divizată problema curentă şi 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* şi *stăpîneşte* este de ordinul $n \log_2 n$ sau $n^2 \log_2 n$, deci polinomial.

Programele elaborate în baza metodei *desparte* şi *stăpîneşte* 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 şi găsirea ei, dacă există, cade în sarcina programatorului.

Întrebări şi exerciții

❶ Explicați schema de calcul a algoritmilor bazați pe metoda *desparte* și *stăpînește*.

❷ Care sînt avantajele şi neajunsurile metodei *desparte* şi *stăpînește*?

❸ Utilizînd metoda *desparte* şi *stăpînește*, elaborați un program care determină suma elementelor mulțimii $A=\{a_1, a_2, ..., a_n\}$ formată din n numere reale.

❹ 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).

❺ Estimați necesarul de memorie și complexitatea temporală a programelor `P155` și `P156`.

❻ 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*.

❼ **Căutarea binară.** Se consideră mulțimea $A=\{a_1, a_2, ..., a_n\}$, 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.

❽ Utilizînd metoda *desparte* și *stăpînește*, elaborați un program care determină cel mai mare divizor comun al numerelor naturale $a_1, a_2, ..., a_n$.

❾ **Sortarea prin interclasare.** Elaborați un program care sortează elementele șirului ($a_1, a_2, ..., a_n$) 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).

❿ **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.

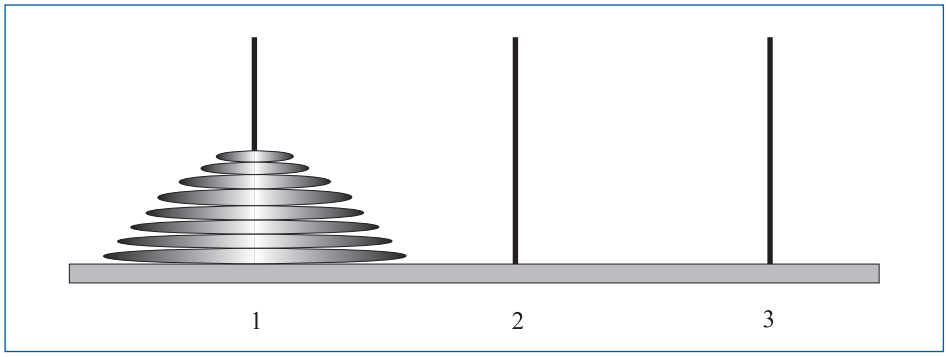

Fig. 5.8. Problema turnurilor din Hanoi

* 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*∈{1, 2, 3}, *i*≠*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 şir de decizii (d_1, d_2, ..., d_p, ..., 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 fişier pe disc etc. Pentru aplicarea acestei metode, este necesar ca problema de rezolvat să satisfacă **principiul optimalității**: dacă (d_1, d_2, ..., d_p, d_{p+1}, ..., d_q) este un şir optim de decizii, atunci şirurile (d_1, d_2, ..., d_p) şi (d_{p+1}, ..., d_q) trebuie să fie optimale.

De exemplu, dacă drumul cel mai scurt dintre *Chişinău* şi *Bălți* trece prin *Orhei*, atunci ambele porțiuni din acest drum – *Chişinău–Orhei* şi *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 utilizînd 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}, ..., d_q. Spunem în acest caz că se aplică **metoda înainte**. Deciziile vor fi luate în ordinea d_q, d_{q-1}, ..., d_1.

2) Fiecare decizie d_p depinde de deciziile d_1, ..., d_{p-1}. În acest caz spunem că se aplică **metoda înapoi**. Deciziile vor fi luate în ordinea d_1, d_2, ..., d_q.

Evident, pentru fiecare problemă propusă, programatorul va verifica mai întîi respectarea principiului optimalității şi 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*×*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*), $1 \le i \le n$, $1 \le j \le m$, 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 mişca 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.

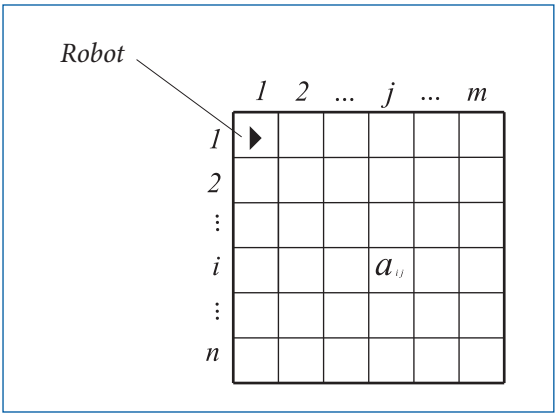

Fig. 5.9. Zonele unui teren aurifer

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 =

	1	2	3	4	5
1	1	2	3	4	5
2	3	4	6	4	6
3	6	2	7	7	5
4	7	2	3	4	5

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 deplasîndu-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 $(i, j\text{-}1)$ sau prin cea din nord cu coordonatele $(i\text{-}1, j)$. Prin urmare, cantitatea maximă de aur pe care o poate extrage robotul ajungînd 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, şi 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*.

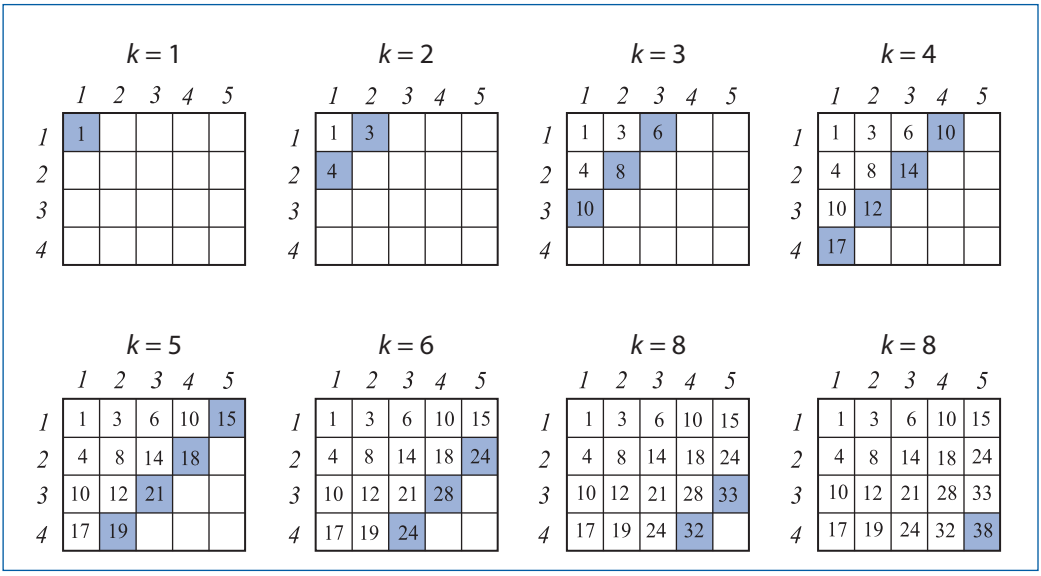

Fig. 5.10. Calcularea elementelor tabloului *C*

Pentru a evita verificările legate de calcularea indicilor *i* şi *j*, în programul ce urmează elementele c_{ij} din prima linie şi 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 bazaţi pe metoda programării dinamice depinde de natura relaţiilor de recurenţă şi în majoritatea cazurilor este polinomială. De exemplu, programul P157 are complexitatea $O(n^3)$.

Întrebări şi exerciţii

❶ Explicaţi esenţa principiului optimalităţii.

❷ În ce ordine pot fi luate deciziile în procesul soluţionării unei probleme prin metoda programării dinamice?

❸ Explicaţi ordinea de luare a deciziilor în problema deplasării robotului pe un teren aurifer.

❹ Demonstraţi că problema deplasării robotului pe un teren aurifer satisface principiul optimalităţii.

❺ Estimaţi necesarul de memorie şi timpul de execuţie al programului `P157`.

❻ Cum credeţi, care sînt avantajele şi neajunsurile algoritmilor bazaţi pe metoda programării dinamice?

❼ **Problema discretă a rucsacului.** Se consideră n obiecte. Pentru fiecare obiect i (i=1, 2, ..., n) se cunoaşte greutatea g_i şi cîştigul c_i care se obţine în urma transportului său la destinaţie. O persoană are un rucsac cu care pot fi transportate unul sau mai multe obiecte greutatea sumară a cărora nu depăşeşte valoarea G_{max}. Elaboraţi un program care determină ce obiecte trebuie să transporte persoana în aşa fel încît cîştigul să fie maxim. Obiectele respective nu pot fi tăiate în fragmente mai mici.

❽ **Drumul de cost minim.** Se consideră o reţea formată din n oraşe. Între unele oraşe există curse directe de autobuz. Costul unui bilet la o cursă directă din oraşul i în oraşul j este de c_{ij} lei. Evident, costul unei curse directe întotdeauna este mai mic decît costul unei călătorii cu transbordări: $c_{ij} < c_{ik} + c_{kj}$. Elaboraţi un program care determină costul minim al unei călătorii din oraşul i în oraşul j.

❾ **Arhivarea fişierelor.** E cunoscut faptul că pe suporturile de memorie externă orice fişier este memorat ca o secvenţă de cifre binare. Pentru a economisi spaţiul de memorare, programele de arhivare descompun fişierul iniţial într-o succesiune de cuvinte binare ce aparţin unui dicţionar şi înscriu pe disc numai numerele de ordine ale cuvintelor respective. De exemplu, în cazul unui dicţionar 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 aparținînd 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 $P_1P_2...P_n$ este definit prin coordonatele (x_i, y_i) ale vîrfurilor P_i, i=1, 2, ..., n. Elaborați un program care descompune poligonul $P_1P_2...P_n$ în triunghiuri trasînd în acest scop un set de coarde P_jP_m, $j \neq m$, $j, m \in \{1, 2, ..., n\}$, care nu se intersectează. Se cere ca lungimea totală a coardelor respective să fie minimă.

5.7. Metoda *ramifică* şi *mărgineşte*

În metoda *ramifică* şi *mărgineşte* (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 $S = \{s_1, s_2, ..., s_n\}$ în care se poate afla un sistem, de exemplu, configurațiile pieselor pe o tablă de şah, 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 şah, 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 şi care optimizează un **criteriu de calitate**, de exemplu, numărul de mutări într-o partidă de şah, distanța parcursă de automobil, timpul de scriere–citire a unui fişier 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 şi 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 micşora numărul de noduri vizitate, în metoda ramifică şi mărgineşte 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 numeşte **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.

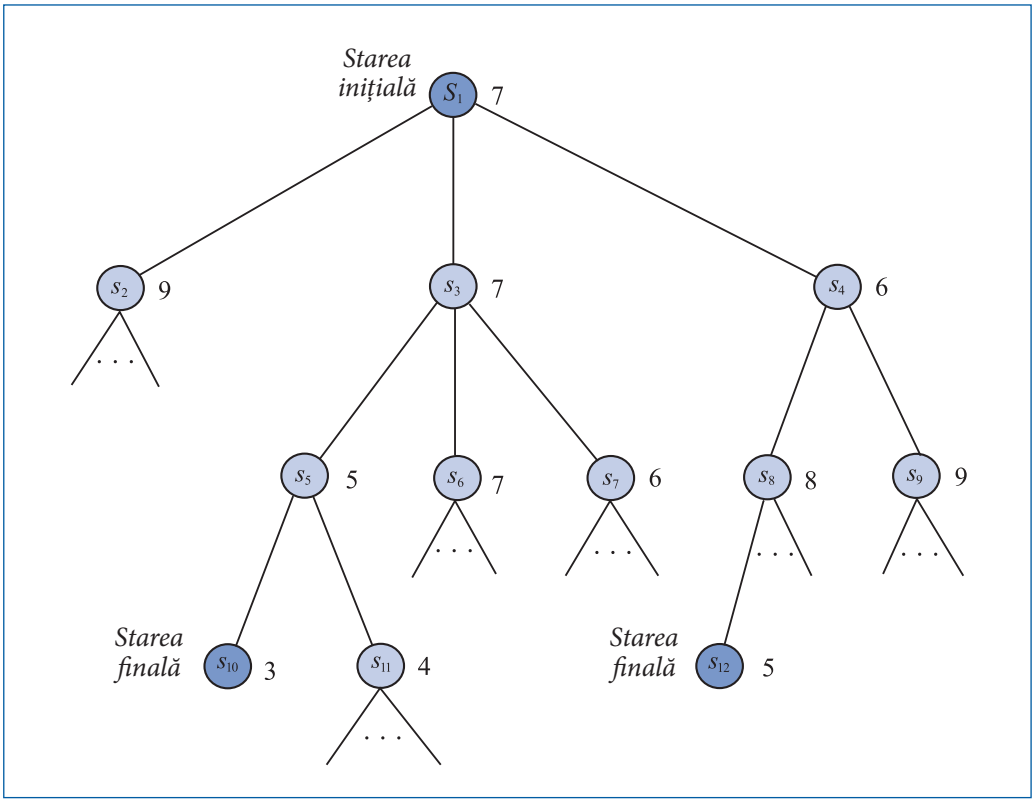

Fig. 5.11. Arborele soluțiilor

Fiind definită o funcție de cost, metoda *ramifică* şi *mărgineşte* 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 şi se generează toți descendenții acestuia.

3. *Etapa mărgineşte*: pentru fiecare descendent se calculează valoarea funcției de cost. Descendenții costul cărora este inaceptabil (de obicei, subarborii respectivi nu conțin stări finale) sînt excluşi din studiu. Descendenții rămaşi se includ în lista nodurilor active.

Punctele 2 şi 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 depăşeşte 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}).

Tabelul 5.2

Parcurgerea de cost minim

Nr. crt.	*Etapa*	*Lista nodurilor active*	*Nodul de cost minim*	*Descendenţii nodului de cost minim*
1.	inițială	(s_1)	-	-
2.	ramifică	$\varnothing$	s_1	s_2, s_3, s_4
3.	mărgineşte	(s_3, s_4)	-	-
4.	ramifică	(s_3)	s_4	s_8, s_9
5.	mărgineşte	(s_3, s_8)	-	-
6.	ramifică	(s_8)	s_3	s_5, s_6, s_7
7.	mărgineşte	(s_8, s_5, s_6, s_7)	-	-
8.	ramifică	(s_8, s_6, s_7)	s_5	s_{10}, s_{11}
9.	mărgineşte	$(s_8, s_6, s_7, s_{10}, s_{11})$	-	-
10.	stop	(s_8, s_6, s_7, s_{11})	s_{10}	-

Schema de calcul a metodei *ramifică* şi *mărgineşte* poate fi redată cu ajutorul următoarei secvențe de instrucțiuni PASCAL:

```
InițializareaListei;
repeat
  Ramifica;
  Margineste;
until Gasit or ListaEsteVida
```

Complexitatea algoritmilor bazați pe metoda *ramifică* şi *mărgineşte* depinde de faptul cum este definită funcția de cost $f: S \to R$. În cazul unor definiții nereuşite 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 reuşită 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:

$$f(s_i) = niv(s_i) + h(s_i),$$

unde $niv(s_i)$ 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 subarborelui de rădăcină s_i.

De exemplu, în cazul unei partide de şah 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* şi *mărgineşte* este posibilă numai atunci cînd valorile $f(s_i)$ ale funcției de cost pot fi calculate fără a construi subarborele 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 bazaţi pe metoda *ramifică* şi *mărgineşte* depinde în mare măsură de experienţa şi iscusinţa programatorului.

Întrebări şi exerciţii

❶ Pentru care tip de probleme poate fi aplicată metoda *ramifică* şi *mărginește*?
❷ Cum se construiește arborele soluțiilor? Ce informație conține fiecare nod al arborelui?
❸ Cum se reprezintă soluția unei probleme în metoda *ramifică* şi *mărginește*?
❹ Care este destinația funcției de cost ? Cum poate fi definită această funcție?
❺ Explicați schema de calcul a metodei *ramifică* şi *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* şi *mărginește*.
⓫ Încercaţi să desenaţi nodurile de pe nivelele 0, 1 şi 2 ale unui arbore ce reprezintă jocul dvs. preferat, de exemplu, lupta maritimă, şah, dame etc.
⓬ Cum credeţi, care sînt avantajele şi neajunsurile algoritmilor bazaţi pe metoda *ramifică* şi *mărginește*?

5.8. Aplicaţiile metodei *ramifică* şi *mărgineşte*

Metoda *ramifică* şi *mărgineşte* poate fi aplicată numai problemelor din enunţul cărora rezultă regulile sau operaţiile necesare pentru generarea directă a arborelui soluţiilor. De obicei, în astfel de probleme se simulează comportamentul unor parteneri ce au scopuri sau tendinţe contrare: aplicaţii militare, scenarii de dezvoltare economică în condiţii de concurenţă, jocuri de şah sau de dame etc. Menţionăm că existenţa unor reguli de generare directă a descendenţilor face inutilă construirea completă a arborelui soluţiilor, micşorîndu-se astfel volumul de memorie internă cerut de algoritm.

Implementarea practică a metodei *ramifică* şi *mărgineşte* presupune rezolvarea următoarelor probleme:

– determinarea mulţimii de stări care, în funcţie 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 şi 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* şi *mărgineşte* în cazul **jocului Perspico**, cunoscut şi sub denumirea **jocul 15**. În acest joc sînt 15 plăcuțe pătrate numerotate de la 1 la 15 (*fig. 5.12*).

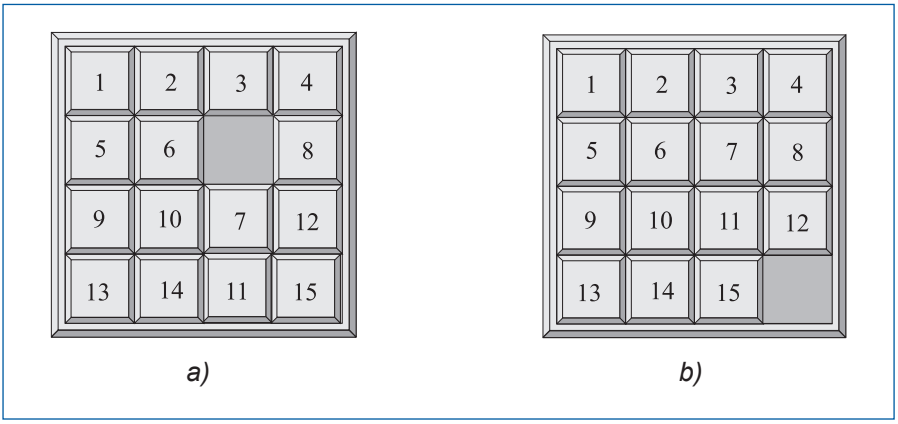

Fig. 5.12. Jocul Perspico:
a – starea inițială; *b* – starea finală

Plăcuțele sînt încadrate într-o ramă pătrată de dimensiunile 4×4, 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 şi 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 şi regulile de trecere dintr-o stare în alta, putem construi **arborele soluțiilor**, o parte din care este prezentată în *figura 5.13*.

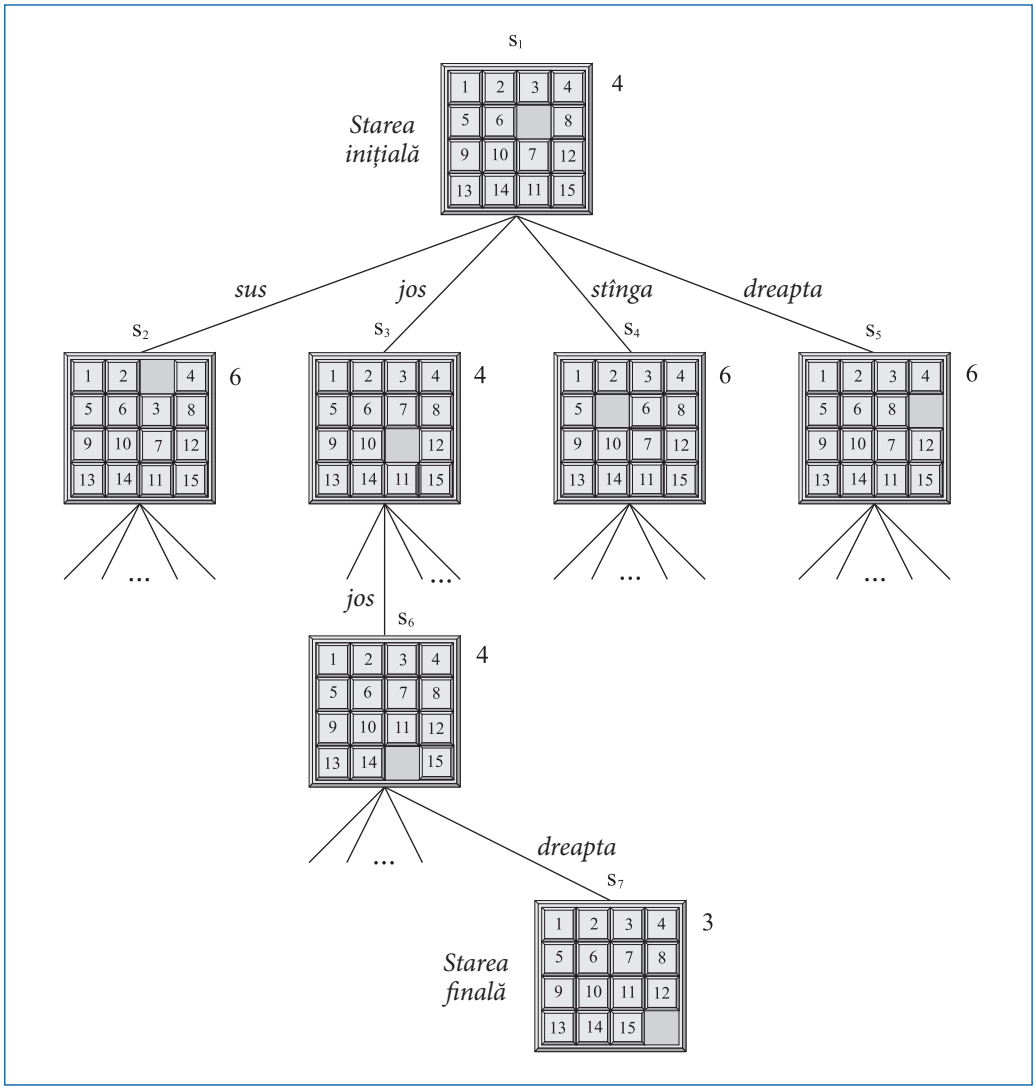

Fig. 5.13. Arborele soluțiilor în jocul Perspico

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 şirului de mutări permise care transformă starea inițială în stare finală.

Funcția de cost $f : S \to R$ 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 întîi subarborele de rădăcină s_i, vom folosi o aproximare prin lipsă a numărului necesar de mutări:

$h(s_i)$ = numărul de plăcuțe care nu se află la locul lor.

De exemplu, pentru arborele din *figura 5.13* avem:

$$f(s_1) = niv(s_1) + h(s_1) = 0 + 4 = 4;$$
$$f(s_2) = niv(s_2) + h(s_2) = 1 + 5 = 6;$$
$$f(s_3) = niv(s_3) + h(s_3) = 1 + 3 = 4;$$
$$f(s_6) = niv(s_6) + h(s_6) = 2 + 2 = 4;$$
$$f(s_7) = niv(s_7) + h(s_7) = 3 + 0 = 3.$$

Arborele soluțiilor şi 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 : boolen;
             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 cîmpurile informaționale `S`, `Nivel`, `Cost`, `Drum` şi cîmpurile de legătură `D`, `Tata`. Cîmpul `Drum` se foloseşte pentru a marca nodurile ce aparțin drumului care leagă nodul rădăcină cu nodul ce conține starea finală. Cîmpul `Tata` se utilizează pentru a memora adresa nodului tată.

Fiecare celulă a listei nodurilor active conține cîmpul informațional `ReferintaNod` în care se indică adresa nodului inclus în listă. Cîmpul de legătură `Urm` conține adresa celulei următoare din listă. Prin urmare, lista nodurilor active conține nu nodurile propriu-zise, ci doar adresele lor.

Subprogramele necesare pentru implementarea metodei *ramifică* şi *mărgineşte* 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
                                           descendentilor}
           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 }
 { Inscrie 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);
 { Inscrie in Str toate starile care pot fi obtinute }
 { din starea curenta prin efectuarea unei mutari permise }
label 1;
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 1;
1: 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;
```

```
    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 1;
  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);
1: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 şi starea finală sînt citite din fişierul `FINPUT.TXT`, iar nodurile extrase la fiecare pas din listă şi drumul găsit sînt înscrise în fişierul `FOUTPUT.TXT`. Conținutul acestor fişiere poate fi vizualizat sau modificat cu ajutorul unor editoare simple de text.

Întrebări şi 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 fişierul de ieşire.

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`.

❺ 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?

❻ 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?

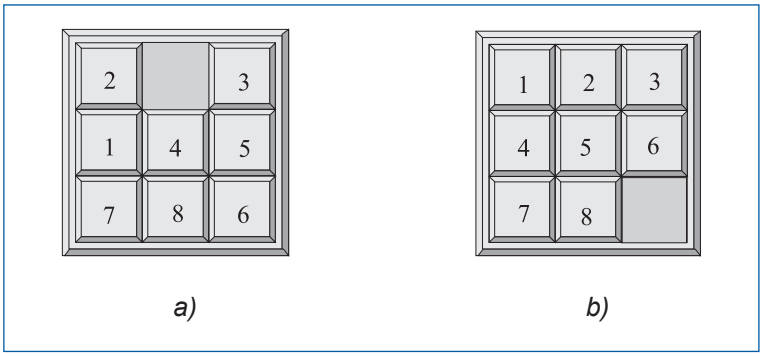

Fig. 5.14. Varianta simplificată a jocului Perspico:
a – starea inițială; *b* – starea finală

❼ Modificați programul `P158` utilizînd 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:

$$g(s_i) = 0 + 0 + 0 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 0 + 0 + 0 + 1 + 3 = 6.$$

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 $h(s_i)$, $g(s_i)$ micșorează timpul de calcul.

❽ Comparați complexitatea programului `P158` cu complexitatea unui program bazat pe metoda trierii.

❾ 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, utilizînd mutările permise, din poziția (α, β) în poziția (γ, δ). Amintim că $\alpha, \gamma \in$ {A, B, ..., H}, iar $\beta, \delta \in$ {1, 2, ..., 8}.

❿ **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, cunoscînd distanțele d_{ij} între orașe, determină ordinea în care ele vor fi vizitate. Se cere ca distanța parcursă de comis-voiajor să fie cît mai mică, iar fiecare oraș să fie vizitat numai o singură dată.

⓫ 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*, ..., *k*, 1) de orașe în care *i*, *j*, ..., $k \in$ {2, 3, ..., *n*}. Pentru a genera toate drumurile posibile, se calculează permutările mulțimii {2, 3, ..., *n*}.

⓬ Efectuați un studiu comparativ al algoritmilor bazați pe metoda *ramifică* și *mărginește* și algoritmilor bazați pe metoda trierii.

5.9. Algoritmi exacți şi algoritmi euristici

Algoritmii utilizați pentru rezolvarea problemelor cu ajutorul calculatorului se împart în două categorii distincte: algoritmi exacți şi algoritmi euristici.

Un algoritm este **exact** numai atunci cînd el găseşte 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 cînd el găseşte 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 şi soluția optimă. Confirmarea exactității acestor algoritmi se reduce la demonstrarea faptului că în procesul derulării pe calculator vor fi generate şi analizate toate elementele din mulțimea soluțiilor posibile. În cazul algoritmilor bazați pe celelalte tehnici de programare – tehnica *Greedy*, metoda reluării, metoda ramifică și mărgineşte 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 nereuşit, soluțiile optime pot fi pierdute şi, în consecință, algoritmul respectiv nu va mai fi exact. Pentru exemplificare vom analiza problema ce urmează.

Problema drumului minim. Se consideră n oraşe legate printr-o rețea de drumuri (*fig. 5.15*). Cunoscînd distanțele d_{ij} dintre oraşele vecine, determinați cel mai scurt drum din oraşul a în oraşul b.

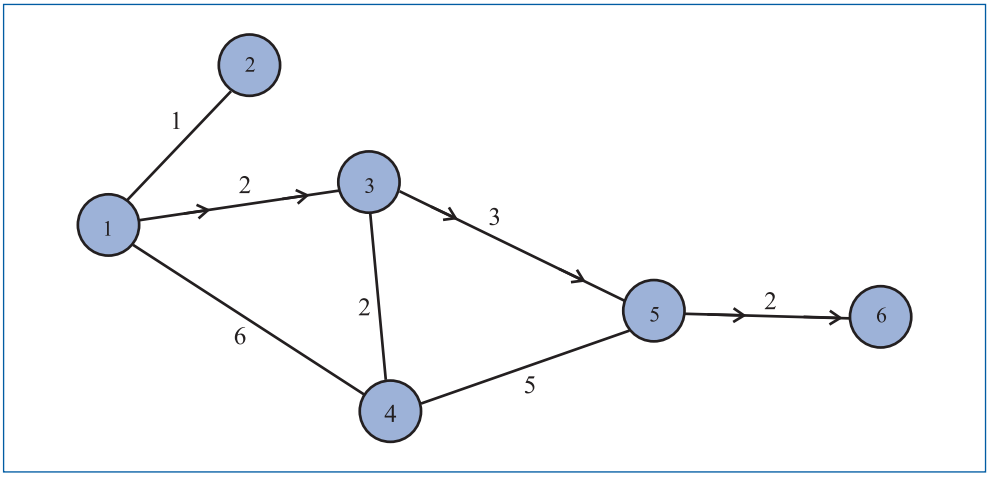

Fig. 5.15. Rețea de drumuri

Datele inițiale ale problemei în studiu pot fi descrise cu ajutorul matricei (tabelului bidimensional) $D = ||d_{ij}||_{n\times n}$ cu n linii şi n coloane, denumită **matricea distanțelor**. În această matrice componenta d_{ij} este egală cu distanța dintre oraşele i, j atunci cînd ele sînt vecine şi 0 în caz contrar. Prin definiție, $d_{ii} = 0$, $i = 1, 2, ..., n$.

De exemplu, pentru oraşele din *figura 5.15* avem:

$D =$

	1	*2*	*3*	*4*	*5*	*6*
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

Drumul minim ce leagă oraşele $a = 1$, $b = 6$ are lungimea 7 şi include localitățile 1, 3, 5, 6.

În general, drumul minim poate fi găsit prin **metoda trierii,** generînd consecutiv toate permutările mulțimilor ordonate

$$X = (a, \underbrace{i, j, \ldots, k}_{q \text{ oraşe}}, b),$$

unde q ia valori de la 0 la n-2. Este clar că algoritmii bazaţi 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 oraşul iniţial $x_1 = a$ şi continuă cu primul dintre vecinii săi nevizitaţi, fie acesta x_2, trecîndu-se la primul dintre vecinii lui x_2 nevizitaţi încă ş.a.m.d. Pentru a construi drumuri cît mai scurte, vom aplica următoarea **regulă intuitivă**: la fiecare pas vom examina, în primul rînd, vecinii nevizitaţi care se află cît mai aproape de oraşul curent.

Drumul în construcţie poate fi reprezentat în forma unui vector:

$$X = (a, x_2, ..., x_{k-1}, x_k, ..., b),$$

în care componenta x_k trebuie să fie unul din vecinii oraşului x_{k-1}. Pentru a sistematiza calculele vom memora vecinii încă nevizitaţi ai oraşului i în mulţimea A_i , i = 1, 2, ..., n. Evident, conform regulii intuitive formulate mai sus, oraşele din fiecare mulţime A_i trebuie sortate în ordinea creşterii distanţelor d_{ij}, $j \in A_i$.

De exemplu, pentru *figura 5.15* iniţial vom avea:

$A_1 = (2, 3, 4);$
$A_2 = (1);$
$A_3 = (1, 4, 5);$
$A_4 = (3, 5, 6);$
$A_5 = (6, 3, 4);$
$A_6 = (5).$

Condiţiile de continuare în metoda reluării rezultă direct din enunţul problemei: oraşul x_k poate fi adăugat la porţiunea de drum deja construită (a, x_2, ..., x_{k-1}) numai atunci cînd:

1) x_k este un vecin încă nevizitat al oraşului x_{k-1}, deci $x_k \in A_{k-1}$;

2) oraşul x_k anterior nu a fost inclus în drumul în curs de construcţie, deci $x_k \neq a$, $x_k \neq x_2$, ..., $x_k \neq x_{k-1}$.

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 şi, evident, nu este un drum minim. Totuşi acest drum este mai bun ca drumul (1, 4, 5, 6) care are lungimea 13.

În programul ce urmează elementele mulţimilor A_1, A_2, ..., A_n sînt plasate la începutul liniilor `A[1]`, `A[2]`, ..., `A[n]` ale tabelului bidimensional `A`, restul poziţiilor avînd valoarea zero.

```
Program P159;
 { Problema drumului minim - metoda reluării }
const nmax=50;
var n : integer; { numărul de oraşe }
    D : array[1..nmax, 1..nmax] of real; { matricea distanţelor }
    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 oraşului k }
var k, i, j, p, q, r : integer;
begin
  for k:=1 to n do
    begin
      { iniţial mulţimea V[k] este vidă }
      for i:=1 to n do V[k,i]:=0;
      { calculăm elementele mulţimii 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 mulţimii 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 1;
var i : integer;
begin
  if MultimeVida(k) then goto 1;
  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);
1: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 oraş. Menționăm că numărul concret de operații efectuate de programul P159 depinde în mare măsură de configurația rețelei de drumuri şi 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 utilizînd 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 oraşele i, j. Componentele c_{ij} se calculează după cum urmează:

1) mai întîi se examinează drumurile minime care leagă oraşele vecine:

$$c_{ij} = \begin{cases} 0, & \text{dacă } i = j; \\ d_{ij}, & \text{dacă oraşele } i, j \text{ sînt vecine;} \\ \infty, & \text{în caz contrar;} \end{cases}$$

2) în continuare se examinează drumurile minime între oraşele i, j formate prin folosirea localităţii k drept punct de conexiune:

$$c_{ij} = \min(c_{ik}, c_{ik}+c_{jk}), \quad i, j \in \{1, 2, ..., n\}, \quad i \neq j, \quad i \neq k, \quad j \neq k;$$

3) punctul 2 se repetă pentru $k = 1, 2, ..., n$.

Valorile curente ale matricei costurilor pentru oraşele din *figura 5.15* sînt prezentate în *figura 5.16*.

Matricea iniţială

	1	2	3	4	5	6
1	0	1	2	6	∞	∞
2	1	0	∞	∞	∞	∞
3	2	∞	0	2	3	∞
4	6	∞	2	0	5	∞
5	∞	∞	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 1$

	1	2	3	4	5	6
1	0	1	2	6	∞	∞
2	1	0	3	7	∞	∞
3	2	3	0	2	3	∞
4	6	7	2	0	5	∞
5	∞	∞	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 2$

	1	2	3	4	5	6
1	0	1	2	6	∞	∞
2	1	0	3	7	∞	∞
3	2	3	0	2	3	∞
4	6	7	2	0	5	∞
5	∞	∞	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 3$

	1	2	3	4	5	6
1	0	1	2	4	5	∞
2	1	0	3	5	6	∞
3	2	3	0	2	3	∞
4	4	5	2	0	5	∞
5	5	6	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 4$

	1	2	3	4	5	6
1	0	1	2	4	5	∞
2	1	0	3	5	6	∞
3	2	3	0	2	3	∞
4	4	5	2	0	5	∞
5	5	6	3	5	0	2
6	∞	∞	∞	∞	2	0

$k = 5$

	1	2	3	4	5	6
1	0	1	2	4	5	7
2	1	0	3	5	6	8
3	2	3	0	2	3	5
4	4	5	2	0	5	7
5	5	6	3	5	0	2
6	7	8	5	7	2	0

$k = 6$

	1	2	3	4	5	6
1	0	1	2	4	5	7
2	1	0	3	5	6	8
3	2	3	0	2	3	5
4	4	5	2	0	5	7
5	5	6	3	5	0	2
6	7	8	5	7	2	0

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 oraşele a, b cu ajutorul tehnicii *Greedy*. Conform acestei tehnici, construcția drumului minim $X = (x_1, ..., x_{k-1}, x_k, ...)$ începe cu oraşul inițial $x_1 = a$. În continuare, la fiecare pas k se alege acel oraş 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 oraşele $a = 1$, $b = 6$ din *figura 5.15* avem:

$X = (1);$
$X = (1, 3);$
$X = (1, 3, 5);$
$X = (1, 3, 5, 6).$

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 construieşte 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 distanţelor }
    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 in V[k] vecinii oraşului k }
var k, i, j, p, q, r : integer;
begin
  for k:=1 to n do
    begin
      { iniţial multimea V[k] este vidă }
      for i:=1 to n do V[k,i]:=0;
      { calculăm elementele mulţimii 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 }
    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; { oraşul candidat la includerea în drumul
                     minim }
    cnd : boolean; { condiţiile de includere a oraşului 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 întîi se încearcă elaborarea unui algoritm exact de complexitate polinomială. Dacă această tentativă eşuează, 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 $X = (a, x_2, ..., x_{k-1}, x_k, ..., b)$ 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, şi anume, orașul x_k trebuie să fie cît mai aproape de orașul x_{k-1}. Evident, formularea condițiilor care acceptă un compromis şi înlocuirea lor cu altele mai simple cade în sarcina programatorului.

Întrebări şi exerciții

❶ Care este diferența dintre algoritmii exacți și cei euristici?

❷ În care cazuri algoritmii euristici sînt „mai buni" decît cei exacți?

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 exacți 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 exacți 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 exacți 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).

Capitolul 6

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:

$$A = \{a_1, a_2, \ldots, a_j, \ldots, a_n\}.$$

Întrucît 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 $\mathbf{A}=(a_1, a_2, \ldots, a_j, \ldots, a_n)$. Evident, o astfel de reprezentare stabileşte implicit o ordine a elementelor mulțimii corespunzătoare ordinii în care apar componentele în vectorul **A**.

Fie A_i o submulțime a lui A. În PASCAL această submulțime poate fi reprezentată prin **vectorul caracteristic al submulțimii**:

$$B_i = (b_1, b_2, \ldots, b_j, \ldots, b_n),$$

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 şi vectorii caracteristici B_i există o corespondență biunivocă:

$$\begin{aligned}
A_1 &= \varnothing && \leftrightarrow B_1 = (0, 0, \ldots, 0);\\
A_2 &= \{a_1\} && \leftrightarrow B_2 = (1, 0, \ldots, 0);\\
A_3 &= \{a_2\} && \leftrightarrow B_3 = (0, 1, \ldots, 0);\\
A_4 &= \{a_1, a_2\} && \leftrightarrow B_3 = (1, 1, \ldots, 0);\\
&\ldots && \ldots\\
A_k &= \{a_1, a_2, \ldots, a_n\} && \leftrightarrow B_k = (1, 1, \ldots, 1);
\end{aligned}$$

Evident, numărul tuturor submulțimilor unei mulțimi este $k = 2^n$.

Reprezentarea submulțimilor cu ajutorul vectorilor caracteristici presupune elaborarea unei unități de program ce conține cîte 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 algoritmii 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 $A=\{a_1, a_2, \ldots, a_n\}$ 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 $\varnothing$, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$ ş.a.m.d. pot fi generate formînd consecutiv vectorii binari $B_1, B_2, \ldots, B_k$.

```
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 intregi:');
  for j:=1 to n do read(A[j]);
  write('Dați m='); readln(m);
  CautareSubmultimi;
  writeln('Sfîrşit');
  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 şi t, variabila t reprezentînd transportul. Valoarea $t = 1$ a transportului din rangul n indică faptul că de la vectorul final $B_k = (1, 1, \ldots, 1)$ se trece la vectorul inițial $B_1 = (0, 0, \ldots, 0)$.

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 $A_1, A_2, \ldots, A_n$, fiecare mulțime A_j fiind formată din m_j numere întregi. Selectați din fiecare mulțime A_j cîte un număr a_j în aşa mod, încît produsul $a_1 \times a_2 \times \ldots \times a_n$ să fie maxim.

Rezolvare. Mulțimea soluțiilor posibile $S=A_1 \times A_2 \times \ldots \times A_n$. Evident, numărul soluțiilor posibile $k = m_1 \times m_2 \times \ldots \times m_n$. Fiecare element s_i al produsului cartezian $A_1 \times A_2 \times \ldots \times A_n$ poate fi reprezentat **prin vectorul indiciilor:**

$$C_i = (c_1, c_2, \ldots, c_j, \ldots, c_n),$$

unde c_j este indicele elementului respectiv din mulțimea A_j. De exemplu, pentru

$$\begin{matrix} \textcircled{1}\ \textcircled{2}\ \textcircled{3} & \textcircled{1}\ \textcircled{2} & \textcircled{1}\ \textcircled{2}\ \textcircled{3} \\ A_1 = (-6,\ 2,\ 1); & A_2 = (4,\ 9); & A_3 = (-8,\ 3,\ 5), \end{matrix}$$

obținem:

s_1 = (-6, 4, -8) ↔ C_1 = (①, ①, ①);
s_2 = (2, 4, -8) ↔ C_2 = (②, ①, ①);
s_3 = (1, 4, -8) ↔ C_3 = (③, ①, ①);
s_4 = (-6, 9, -8) ↔ C_4 = (①, ②, ①);
s_5 = (2, 9, -8) ↔ C_5 = (②, ②, ①);
…
s_{18}= (1, 9, 5) ↔ C_{18} = (③, ②, ③),

unde ①, ② şi ③ sînt indicii elementelor din mulțimile respective.

Vectorii C_1, C_2, …, C_k pot fi generați în ordine lexicografică pornind de la vectorul inițial C_1=(1, 1, …, 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 $C_k = (m_1, m_2, \ldots, m_n)$ se trece la vectorul inițial $C_1 = (1, 1, \ldots, 1)$.

Menționăm că dacă numărul de mulțimi n este cunoscut pînă la scrierea programului, generarea elementelor produsului cartezian $A_1 \times A_2 \times \ldots \times A_n$ 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 bazaţi pe generarea tuturor elementelor unui produs cartezian este $O(m^n)$, unde $m = \max(m_1, m_2, \ldots, m_n)$.

Întrebări şi exerciţii

❶ Submulţimile A_i, A_j ale mulţimii A sînt reprezentate prin vectori caracteristici. Elaboraţi procedurile necesare pentru efectuarea următoarelor operaţii: $A_i \cap A_j$, $A_i \cup A_j$, $A_i \setminus A_j$, $\bar{A}_i$.

❷ Orice submulţime A_i, $A_i \subseteq A$, poate fi reprezentată printr-un vector cu n componente, unde elementele submulţimii A_i sînt plasate la începutul vectorului, iar restul poziţiilor au o valoare ce nu aparţine mulţimii A. Elaboraţi procedurile necesare pentru efectuarea operaţiilor frecvent întîlnite în calculul cu mulţimi: $\cup$, $\cap$, $\setminus$, $\bar{}$. Cum credeţi, care reprezentare a submulţimilor este mai comodă: prin vectorii caracteristici sau prin vectorii ce conţin chiar elementele submulţimii?

❸ Estimaţi timpul de execuţie a procedurii `CautareSubmultimi` din programul `P161`. Verificaţi aceste estimări prin măsurări directe ale timpului de execuţie pentru diferite valori ale lui n.

❹ Se consideră mulţimea A formată din primele n caractere ale alfabetului latin. Elaboraţi un program care afişează la ecran toate submulţimile acestei mulţimi.

❺ Se consideră numărul natural n = 32*35*17*, format din 9 cifre zecimale. Determinaţi cifrele care trebuie înscrise în poziţiile marcate cu simbolul * pentru ca numărul obţinut să se împartă fără rest la m.

❻ Estimaţi timpul de execuţie a procedurii `CautareaProdusuluiMaximal` din programul `P162`. Verificaţi aceste estimări prin măsurări directe ale timpului de execuţie pentru diferite valori ale lui n şi $m_1, m_2, \ldots, m_n$.

❼ Într-un coş sînt m mere şi p pere. Să se genereze toate posibilităţile de a alege f fructe dintre care k să fie mere.

❽ Elaboraţi o procedură recursivă care generează toate elementele unui produs cartezian.

❾ Fie $A = (a_1, a_2, \ldots, a_j, \ldots, a_n)$ o mulţime ordonată de caractere numită **alfabet**. Numim **cuvînt de lungime *p*** orice succesiune de p caractere din alfabetul A. Elaboraţi o procedură care generează toate cuvintele de lungimea p.

6.2. Analiza combinatorie

În rezolvarea multor probleme implementarea algoritmilor bazaţi pe analiza consecutivă a soluţiilor posibile presupune generarea permutărilor, aranjamentelor sau combinărilor unei mulţimi.

Generarea permutărilor. E cunoscut faptul că numărul de permutări posibile ale unei mulțimi $A=\{a_1, a_2, ..., a_n\}$ cu n elemente se determină ca $P_n = n!$. Acest număr poate fi calculat cu ajutorul funcției *factorial*, exprimată în formă iterativă:

$$P_n = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$$

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) şi (a_2, a_1). Pentru $A=\{a_1, a_2, a_3\}$ cele $P_3 = 3! = 6$ permutări sînt:

$$(a_1, a_2, a_3); \quad (a_1, a_3, a_2); \quad (a_2, a_1, a_3);$$
$$(a_2, a_3, a_1); \quad (a_3, a_1, a_2); \quad (a_3, a_2, a_1).$$

Întrucît între permutările mulțimii $A=\{a_1, a_2, ..., a_n\}$ şi permutările mulțimii $I=\{1, 2, ..., n\}$ există o corespondență biunivocă, problema generării permutărilor oricărei mulțimi A cu n elemente se reduce la generarea permutărilor mulțimii $\{1, 2, ..., n\}$, 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, şi anume de la **permutarea identică** $(1, 2, ..., n)$.

Avînd construită o permutare $p = (p_1, ..., p_{i-1}, p_i, p_{i+1}, ..., p_n)$, pentru determinarea următoarei permutări p' care îi urmează în ordine lexicografică se caută acel indice i care satisface relațiile:

$$p_i < p_{i+1}; p_{i+1} > p_{i+2} > ... > p_n.$$

În continuare, elementul p_i este înlocuit cu cel mai mic dintre elementele $p_{i+1}, ..., p_n$ care este mai mare decît p_i, fie el p_k:

$$(p_1, ..., p_{i-1}, p_k, p_{i+1}, ..., p_{k-1}, p_i, p_{k+1}, ..., p_n).$$

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 $(n, (n - 1), ..., 1)$ şi algoritmul se termină.

De exemplu, pentru n=3 se obțin permutările:

$$(1, 2, 3); \quad (1, 3, 2); \quad (2, 1, 3);$$
$$(2, 3, 1); \quad (3, 1, 2); \quad (3, 2, 1).$$

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 şi 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, algoritmii 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 $A=\{a_1, a_2, ..., a_n\}$ cu n elemente este dat de formula:

$$A_n^m = \frac{n!}{(n-m)!}$$

sau, pentru a evita folosirea funcției *factorial*,

$$A_n^m = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot (n-m+2) \cdot (n-m+1).$$

Ca şi în cazul permutărilor, putem reduce problema generării aranjamentelor unei mulțimi arbitrare A la generarea aranjamentelor mulțimii $I=\{1, 2, ..., n\}$. De exemplu, pentru $I=\{1, 2, 3\}$ şi $m=2$ cele $A_3^2 = 6$ aranjamente sînt:

(1, 2); (2, 1); (1, 3);
(3, 1); (2, 3); (3, 2).

Generarea aranjamentelor poate fi făcută în ordine lexicografică, pornind de la aranjamentul cel mai mic $a=(1, 2, ..., m)$.

Fie $a=(a_1, a_2, ..., a_i, ..., a_m)$ un aranjament oarecare. Pentru a determina succesorul a' al aranjamentului a, se caută mai întîi 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, ..., n cu care ar putea fi înlocuit a_i este disponibilă. Pentru a putea efectua mai uşor aceste verificări, se utilizează vectorul $D=(d_1, d_2, ..., d_i, ..., d_n)$, 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 $a_i, a_{i+1}, ..., a_m$ 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, ..., *n*), deci procesul generării s-a încheiat. Pentru a semnala 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 1;
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 1;
    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 }
```

```
            goto 1;
          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.
```

Aşadar, complexitatea temporală a algoritmilor bazați pe generarea tuturor aranjamentelor este cel mult O(n!).

Generarea combinărilor. Numărul de combinări de n elemente luate cîte m ($m \le n$) se calculează ca

$$C_n^m = \frac{A_n^m}{P_m} = \frac{n!}{m!(n-m)!}.$$

De exemplu, pentru I={1, 2, 3} şi m=2 avem $C_3^2 = 3$ combinări:

{1, 2}; {1, 3}; {2, 3}.

Generarea combinărilor poate fi făcută în ordine lexicografică, pornind de la combinarea inițială {1, 2, ..., m}.

Fie dată o combinare c={c_1, c_2, ..., c_i, ..., c_m}. Combinarea c' care îi urmează imediat în ordine lexicografică se determină astfel:

– se stabileşte 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 combinarea c'={c_1, ..., c_{i-1}, c_i+1, c_i+2, ..., $c_i+n-i+1$}.

Dacă nu există un indice i care satisface condițiile de mai sus, înseamnă că au fost generate toate combinările.

În programul ce urmează combinările mulțimii I={1, 2, ..., 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 combinărilor este de ordinul $O(n^k)$, unde $k=\min(m, n-m+1)$, deci polinomial.

Întrebări şi exerciții

❶ Scrieți un program PASCAL care afișează la ecran numărul permutărilor P_n, numărul aranjamentelor A_n^m și numărul combinărilor 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, ..., *n*}.

❸ Utilizînd metoda reluării, schițați trei algoritmi pentru generarea, respectiv, a permutărilor, aranjamentelor și combinărilor 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 $A=\{a_1, a_2, ..., a_n\}$. 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 combinărilor posibile?

❾ Există oare tehnici de programare care ar permite evitarea unei analize exhaustive a tuturor permutărilor, aranjamentelor sau combinărilor posibile?

❿ Se consideră mulțimea numerelor întregi $A=\{a_1, a_2, ..., a_n\}$. 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}.

Capitolul 7

PROBLEME RECAPITULATIVE

Problemele ce urmează au fost propuse la diverse concursuri de informatică. Rezolvarea lor necesită cunoaşterea profundă a tehnicilor de programare şi a metodelor de estimare a complexităţii algoritmilor.

1. Cercuri. Un pătrat cu latura a cm conţine n cercuri ($n \le 100$). Fiecare cerc i este definit prin coordonatele centrului (x_i, y_i) şi raza r_i. Elaboraţi un program care în cel mult t secunde calculează cît mai exact aria obţinută prin reuniunea celor n cercuri.

2. Puncte. Se dau n puncte în plan, $n \le 100$. 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 depăşeşte g_{ij} tone. Determinaţi 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 aceeaşi foaie se inserează o linie vidă. Fragmentarea textelor este interzisă, adică toate liniile oricărui text trebuie imprimate pe aceeaşi foaie. Elaboraţi un program care determină numărul minim de foi necesare pentru a tipări toate textele.

5. Circumferinţe. Se consideră n puncte pe un plan cartezian. Fiecare punct i este definit prin coordonatele sale x_i, y_i. Elaboraţi un program care verifică dacă punctele în studiu aparţin unei circumferinţe.

6. Reţeaua telefonică. Se consideră n oraşe, $n \le 100$, care trebuie legate prin cabluri într-o reţea telefonică. Pentru fiecare oraş i sînt cunoscute coordonatele carteziene x_i, y_i. Cablurile telefonice ce leagă oricare două oraşe nu pot avea ramificaţii. Abonaţii reţelei telefonice comunică între ei direct sau prin intermediul staţiilor telefonice din alte oraşe. Lungimea cablului care leagă oraşele i, j este egală cu distanţa dintre ele. Determinaţi lungimea sumară minimă a cablurilor necesare pentru a conecta toate oraşele.

7. Evaluarea expresiilor. Se consideră expresiile aritmetice formate din numere întregi, parantezele `(`, `)` şi operatorii binari `+`, `-`, `*`, **`mod`**, **`div`**. Scrieţi un program care evaluează expresiile aritmetice în studiu.

8. Psihologie. Se consideră n angajaţi, $n \le 100$, care trebuie repartizaţi în m echipe. Fiecare echipă este formată din k angajaţi, $k \cdot m = n$. Relaţia de compatibilitate între angajaţii 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 coeficienţii r_{ij} pentru toate perechile posibile (i, j) din cadrul fiecărei echipe. Determinaţi 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. Elaboraţi un program care verifică dacă se poate desena o circumferinţă cu centrul în unul din punctele în studiu şi care ar trece prin toate celelalte puncte.

10. Intersecţia dreptunghiurilor. Se dau n dreptunghiuri ($n \le 10$) care au laturile paralele cu axele de coordonate, iar coordonatele vîrfurilor sînt numere naturale din mulţimea {0, 1, 2, ..., 20}. Elaboraţi un program care calculează aria figurii obţinute prin intersecţia celor n dreptunghiuri.

11. Reuniunea dreptunghiurilor. Se dau n dreptunghiuri ($n \le 10$) care au laturile paralele cu axele de coordonate, iar coordonatele vîrfurilor sînt numere reale. Elaboraţi un program care calculează aria figurii obţinute prin reuniunea celor n dreptunghiuri.

12. Numere prime. Calculaţi 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 ... P_nP_1, $n \le 20$, care nu se autointersectează. În punctul A din interiorul liniei este situat un observator. Pentru observator unele segmente ale liniei frînte pot fi invizibile. Elaboraţi un program care calculează numărul segmentelor invizibile.

14. Felinare. Un parc de formă dreptunghiulară este împărţit în pătrate de aceleaşi 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 şi a celor opt pătrate vecine. Elaboraţi 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 şi 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 colţului stînga-jos al pătratului respectiv.

Pentru operaţiile de tăiere se foloseşte un strung cu laser. Fiecare operaţie de tăiere include:

– fixarea unei plăci pe masa de tăiere;

– stabilirea puterii laserului în funcţie 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 operaţii de tăiere se determină după formula $c = d_{\max}$, unde $d_{\max}$ este densitatea maximă a bucăţilor 1×1 peste marginile cărora trece raza laserului. Evident, costul total T poate fi determinat adunînd costurile individuale c ale tuturor operaţiilor de tăiere necesare pentru obţinerea bucăţilor 1×1. Scrieţi un program care calculează costul minim T.

16. Județe. Teritoriul unei țări este împărțit în județe (*fig. 7.1*). Pe hartă, frontiera țării şi granițele administrative ale fiecărui județ reprezintă cîte 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.

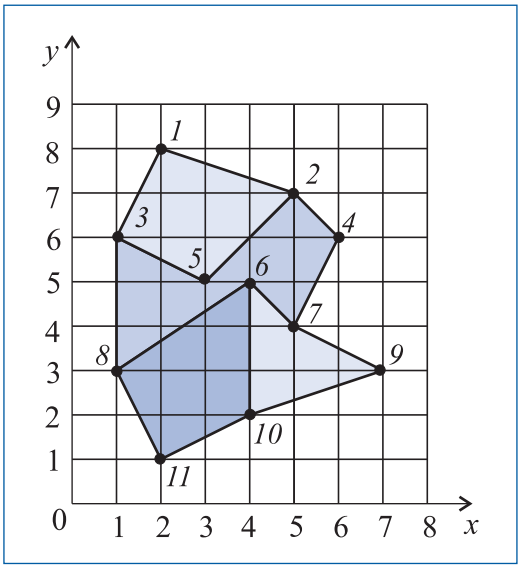

Fig. 7.1. Județele unei țări

Un virus de calculator a distrus parțial informația despre granițele administrative, lăsînd intacte următoarele date:

– numărul n şi coordonatele (x_i, y_i) ale tuturor vîrfurilor de poligoane;

– numărul de segmente m care formează laturi de poligoane şi informația despre extremitățile fiecărui segment.

Scrieți un program care determină numărul de județe d şi 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 ştie că are grosimea h_i şi 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. Fişierul text `TURNURI.IN` conține pe prima linie numărul n. Pe fiecare din următoarele n linii se conțin cîte trei numere întregi pozitive x_i, y_i, h_i separate prin spațiu.

Date de ieşire. Fişierul text `TURNURI.OUT` va conține pe o singură linie un număr întreg – înălțimea maximă a turnului.

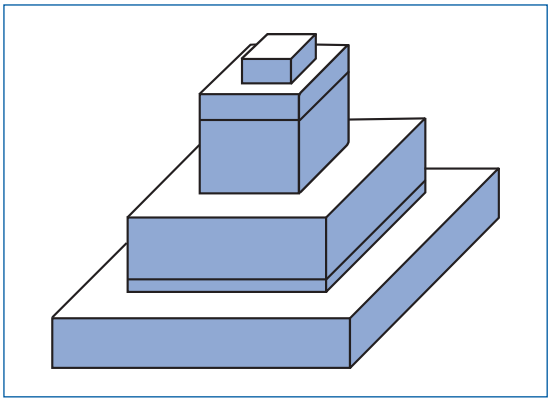

Fig. 7.2. Turn construit din plăci dreptunghiulare

Exemplu:

TURNURI.IN

```
5
3 4 2
3 4 3
4 3 2
1 5 4
2 2 1
```

TURNURI.OUT

```
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
```

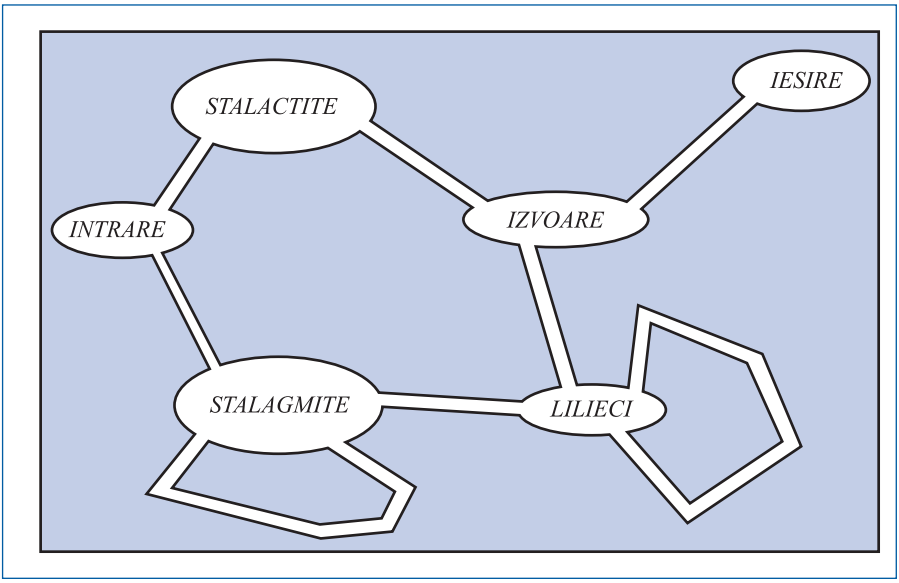

Fig. 7.3. Planul unei peşteri

Trecerea speologului din peştera curentă în peştera 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 ieşire. Fişierul 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 peşteră ocupă o linie separată. Dacă un astfel de drum nu există, fişierul 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 depăşi 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 creşte o singură specie de arbori. Orice specie de arbori poate ocupa una sau mai multe zone, nu neapărat vecine. Elaborați un program care găseşte 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, cunoaşte o ştire foarte importantă pe care el o comunică prietenilor săi. În continuare, orice participant j, care deja cunoaşte această ştire, o comunică, de asemenea, numai prietenilor săi. Elaborați un program care determină numărul de participanți p care vor afla ştirea respectivă. Relația de prietenie este definită prin m perechi distincte de tipul $\{j, k\}$ cu semnificația „participanții j, k sînt prieteni". Se consideră că $3 \le n \le 1000$ şi $2 \le m \le 30000$.

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 şi $m_i = 0$ 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 soluțe şi $3 \le n \le 10000$.

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 şir de numere de cutii $i_1, i_2, i_3, ..., i_k$ 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 şir.

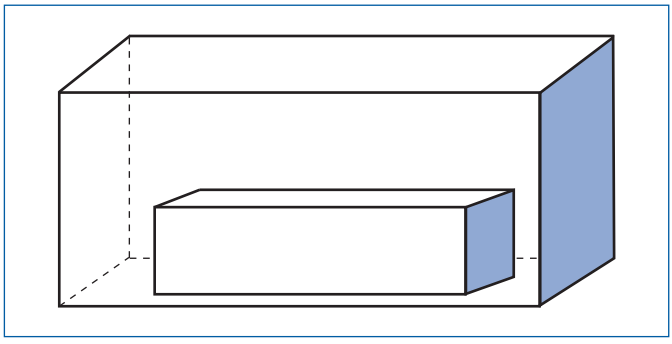

Fig. 7.4. Cutii imbricate

Date de intrare. Fişierul text `CUTII.IN` conţine pe prima linie numărul natural n. Fiecare din următoarele n linii conţine cîte trei numere naturale x_i, y_i, z_i separate prin spaţiu.

Date de ieşire. Fişierul text `CUTII.OUT` va conţine 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
```

Restricţii: $2 \le n \le 500$; $1 \le x_i, y_i, z_i \le 30\,000$. Timpul de execuţie nu va depăşi 2 secunde.

Bibliografie

1. *Cerchez Emanuela, Şerban Marinel*. Informatică. Manual pentru clasa a X-a. Filiera teoretică, profilul matematică-informatică. Iaşi: Editura POLIROM, 2000, 200 p.

2. *Cerchez Emanuela*. Informatică. Culegere de probleme pentru liceu. Iaşi, Editura POLIROM, 2002, 240 p.

3. *Galatan Suzana, Ghinea Diana, Întuneric Ana, Radu Stefana*. Ghid de pregătire pentru BAC. Informatică. Intensiv. Pascal C/C++. Bucureşti, 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. Iaşi, Editura POLIROM, 2004, 456 p.

6. *Gremalschi Anatol, Mocanu Iurie, Spinei Ion*. Informatică. Limbajul PASCAL. Manual pentru clasele IX-XI. Chişinău, Editura Ştiinţa, 2003, 256 p.

7. *Ivaşc Cornelia., Prună Mona*. Bazele informaticii (Grafuri şi elemente de combinatorică). Proiect de manual pentru clasa a X-a. Profil informatică. Bucureşti, Editura Petrion, 1995, 175 p.

8. *Livovschi Leon, Georgescu Horia*. Sinteza şi analiza algoritmilor. Bucureşti, Editura Ştiinţifică şi Pedagogică, 1986, 458 p.

9. *Moraru Florin*. Bacalaureat. Informatică. Bucureşti, Editura Petrion, 2000, 319 p.

10. *Negreanu Dan*. Probleme de matematică rezolvate cu calculatorul. Bucureşti, Editura Teora, 1998, 214 p.

11. *Roşca Ion Gh., Cocianu Cătălina, Uscatu Cristian*. Bazele informaticii. Manual pentru clasa a X-a, licee teoretice. Bucureşti, Editura ALL EDUCAȚIONAL, 1999, 64 p.

12. *Roşca Ion Gh., State Luminița, Ghilic-Micu Bogdan, Cocianu Cătălina-Luca, Stoica Marian, Uscatu Cristian*. Informatică. Manual pentru clasa a 10-a. Profilul matematică-informatică. Bucureşti, ALL EDUCAȚIONAL, 2000, 96 p.

13. *Sorin Tudor*. Informatică (Tehnici de programare). Manual pentru clasa a X-a. Varianta Pascal. Bucureşti, 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., Спиней 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