

5

ALGORITMI RECURSIVI METODELE DIVIDE ET IMPERA ȘI BACKTRACKING

Recursivitatea este o tehnică de programare bazată pe apelarea unui subprogram de către el însuși. În cadrul capitolului sunt prezentate calculul recursiv, metoda „divide et impera” și metoda backtracking implementată recursiv.

5.1 Calcul recursiv

Calculul valorii $n!$ pentru n dat poate fi efectuat pe baza formulei $n! = n(n-1)!$, pentru $n \geq 1$ și $0! = 1$. Dacă $Fact(n)$ este funcția Pascal care calculează $n!$, atunci, dacă $n \geq 1$ evaluarea lui $Fact(n)$ rezultă prin multiplicarea cu n a valorii calculate de apelul $Fact(n-1)$, cu $Fact(0) = 1$. Cu alte cuvinte, apelul funcției $Fact(n)$ realizează calculul “imediat” dacă $n=0$, altfel presupune un nou apel al aceleiași funcții pentru valoarea argumentului egală cu $n-1$. Cazurile în care este posibilă evaluarea “imediată” se numesc *condiții terminale*.

În Pascal, funcția $Fact$, este:

```
function Fact(n:byte):word;  
begin  
  if n=0 then Fact:=1  
  else Fact:=n*Fact(n-1);  
end;
```

Utilizarea formulei $C_n^k = \frac{n!}{k!(n-k)!}$ pentru calculul combinațiilor (n, k date)

ridică dificultăți deoarece $n!$, pentru $n \geq 13$, nu poate fi reprezentat în calculator ca dată de un tip întreg, chiar dacă numărul C_n^k este relativ mic și poate fi reprezentat ca întreg. Pe baza relației de recurență $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ rezultă ceea ce este cunoscut sub numele de triunghiul lui Pascal.

De exemplu, triunghiul lui Pascal, pentru $n = 7$, este:

				1					
				1		1			
			1		2		1		
		1		3		3		1	
	1		4		6		4		1
	1	5		10		10		5	1
	1	6	15		20		15	6	1
1	7	21	35		35	21	7		1

Secvența de program pentru calculul C_n^k , $0 \leq k \leq n \leq 13$ este:

```

var
n,i,j:word;
  x:array[0..13,0..13] of word;
begin
write('Valoarea pentru n=');
readln(n);
if n>13 then
  writeln('Eroare')
else
begin
x[0,0]:=1;
for i:=1 to n do
  begin
    x[i,0]:=1;
    x[i,i]:=1;
  end;
for i:=1 to n-1 do
  for j:=1 to i do
    x[i+1,j]= x[i,j-1]+x[i,j];
  end;
for i:=0 to n do
  begin
    for j:=0 to i do write(x[i,j], ' ');
    writeln;
  end;
end.

```

Valorile combinațiilor sunt calculate în componentele tabeli x, fiind utilizată numai $\frac{(n+1)(n+2)}{2}$ din cele 14^2 celule de memorie rezervate.

Se poate proceda, însă, altfel. Se presupune că *function comb(n,k)* calculează C_n^k . Conform relației de recurență, dacă $n \geq k \geq 1$, atunci evaluarea corespunzătoare apelului *comb(n,k)* revine la însumarea rezultatelor obținute prin apelurile *comb(n-1,k)* și *comb(n-1, k-1)*, unde *comb(n,0)=1*, $n \geq 0$. Dacă evaluările *comb(n-1,k)* și *comb(n-1, k-1)* sunt realizate în același mod, rezultă că apelul *comb(n,k)* va determina o secvență de apeluri ale aceleiași funcții pentru valori ale argumentelor din ce în ce mai mici, până când este îndeplinită una din condițiile terminale *comb(n,0)=1*, *comb(k,k)=1*.

Soluția recursivă a evaluării C_n^k este:

```
function comb(n,k:byte):word;
begin
  if k>n then comb:=0
  else
    if (k=0) or (k=n) then comb:=1
    else comb:=comb(n-1,k)+comb(n-1,k-1);
  end;
```

Un alt exemplu este următorul. Se presupune că f_0, f_1, α, β sunt numere reale date și pentru $p \geq 2$, $f_p = \alpha f_{p-1} + \beta f_{p-2}$. Se cere să se calculeze f_n pentru n dat. Șirul definit de relația precedentă pentru $f_0=f_1=\alpha=\beta=1$ se numește *șirul lui Fibonacci*. Dacă *Fib(n)* este funcția Pascal care calculează cel de-al n -lea element din șirul considerat, atunci evaluarea lui *Fib(n)* revine la însumarea valorilor lui *Fib(n-1)* și *Fib(n-2)* ponderate de constantele α și β , adică rezolvarea problemei *Fib(n)* poate fi redusă la rezolvarea problemelor *Fib(n-1)*, *Fib(n-2)* cu condițiile terminale *Fib(0)=f₀*, *Fib(1)=f₁*.

Funcția Pascal *Fib* pentru calculul celui de-al n -lea termen al șirului definit anterior este:

```
function Fib(n:word;alfa, beta, f0,f1:real):real;
begin
  if n=0 then Fib:=f0
  else if n=1 then Fib:=f1
  else Fib:=alfa*Fib(n-1)+beta*Fib(n-2);
end;
```

Fiecare apel *Fib(n)* pentru $n > 1$ determină încă două apeluri: *Fib(n-1)* și *Fib(n-2)*. Numărul total de apeluri efectuate până la rezolvarea problemei *Fib(n)* crește exponențial, în funcție de parametrul n . Mai mult, rezolvarea problemei *Fib(n-2)* are loc atât la *apelul Fib(n-1)*, cât și la *apelul determinat de Fib(n)*. Datorită acestor inconveniente, este preferată o soluție iterativă pentru calculul unui termen de rang dat al șirului lui Fibonacci.

Se spune că astfel de apeluri sunt *recursive directe*. Schema unui apel recursiv poate fi descrisă în modul următor. Se verifică dacă este îndeplinită cel puțin una din condițiile terminale. Dacă este îndeplinită o condiție terminală, atunci calculul se încheie și se revine la unitatea apelantă. În caz contrar, este inițiat calculul pentru noile valori ale parametrilor, calcul care presupune unul sau mai multe apeluri recursive.

Mecanismul prin care este efectuat apelul unui subprogram se bazează pe utilizarea stivei memoriei calculatorului. Fiecare apel determină introducerea în stivă (operația *push*) a valorilor/adreselor parametrilor formali, adresei de revenire și a variabilelor locale. La momentul execuției, aceste informații sunt extrase cu eliminare din stivă (operația *pop*), eliberându-se spațiul ocupat.

În cazul subprogramelor recursive, mecanismul funcționează astfel: se generează un număr de apeluri succesive cu ocuparea spațiului din stivă necesar efectuării acestor apeluri până la îndeplinirea unei condiții terminale; apelurile sunt executate în ordinea inversă celei în care au fost generate, iar operația *push* poate produce depășirea spațiului de memorie rezervat în stivă.

Astfel, în cazul apelului *Fact(3)*, secvența de apeluri recursive inițiate este: *Fact(2)*, *Fact(1)*, *Fact(0)*. În continuare execuția determină *Fact(0)=1*, *Fact(1)=1*Fact(0)=1*, *Fact(2)=2*Fact(1)=2*, *Fact(3)=3*Fact(2)=6*. Evoluția determinată de apelul *Fact(3)* în stivă este ilustrată în figurile 5.1 și 5.2, unde (○) reprezintă adresa de revenire în punctul de unde a fost efectuat apelul *Fact(3)*.

Apelurile recursive ale unei proceduri sau funcții pot fi și indirecte, în sensul că este efectuat un apel al unei alte proceduri sau funcții care, la rândul ei, inițiază un apel al procedurii sau funcției inițiale.

Un exemplu simplu îl reprezintă calculul valorilor funcției $h=f \circ g \circ f$, unde $f, g: \mathbf{R} \rightarrow \mathbf{R}$ sunt funcții date. Un mod de a rezolva această problemă este descris în continuare. Dacă $f(x)$, $g(x)$ sunt funcțiile Pascal care descriu calculul necesar evaluării funcțiilor date în punctul x , atunci pentru calculul valorii $h(x)$ este necesar apelul funcției f , urmat de apelul funcției g care, la rândul ei, apelează din nou funcția f .

Pentru funcțiile f, g definite prin

$$f(x) = \begin{cases} 2x + 1, & x < 3 \\ x^2 + 2, & x \geq 3 \end{cases}, \quad g(x) = \begin{cases} x^2 - 3x + 2, & x \leq 1 \\ 3x + 5, & x > 1 \end{cases}$$

funcția Pascal $h(x)$ este:

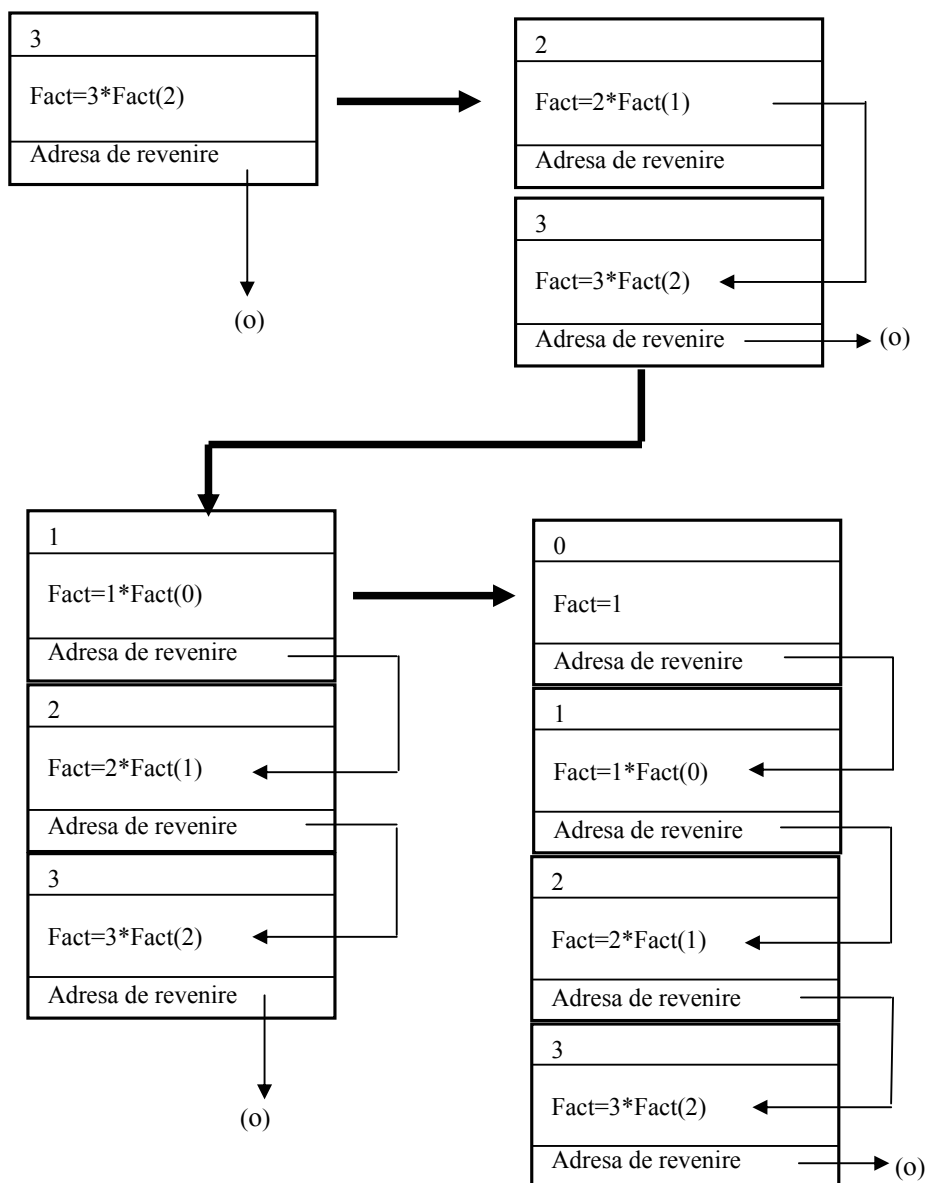


Fig. 5.1 Evoluția în stivă până la condiția terminală `Fact(0):=1`

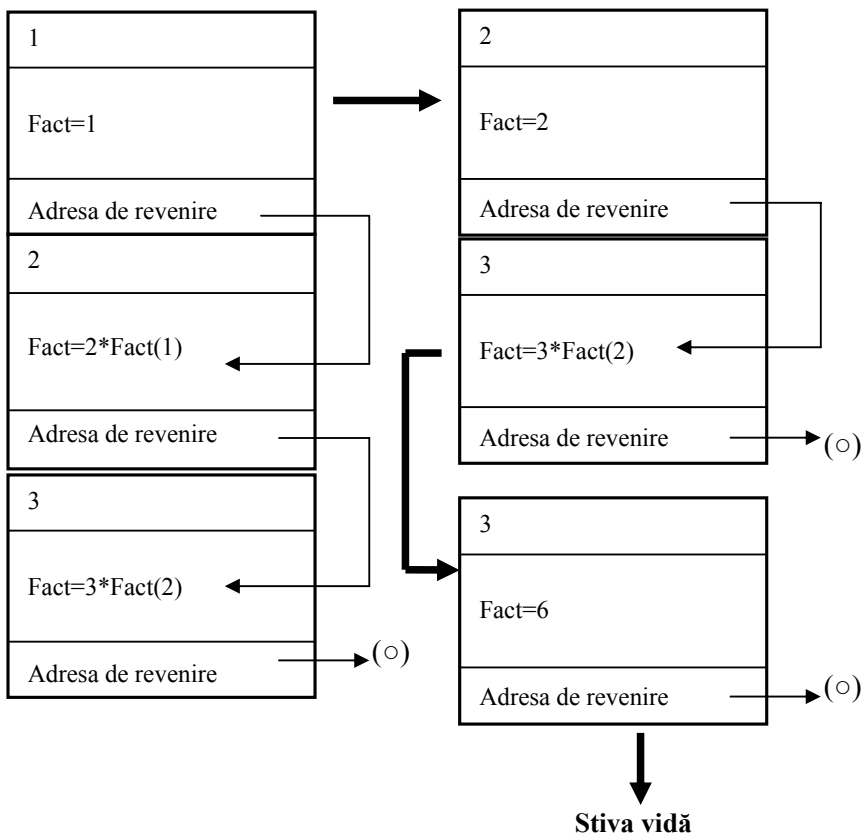


Fig. 5.2 Eliberarea stivei după execuția determinată de condiția terminală

```
function f(x:real):real;
begin
  if x<3 then f:=2*x+1
  else f:=x*x+2
end;

function g(x:real):real;
begin
  if x<=1 then g:=x*x-3*x+2
  else g:=3*x+5
end;

function h(x:real):real;
begin
  h:=f(g(f(x)));
end;
```

5.2 Metoda “divide et impera”

Metoda “divide et impera” presupune descompunerea problemei de rezolvat în două sau mai multe subprobleme (probleme “mai simple”), determinarea soluțiilor acestora care, apoi, compuse după reguli simple, furnizează soluția problemei inițiale.

De exemplu, se presupune că se dorește aflarea valorii maxime dintr-o secvență $\{a_1, \dots, a_n\}$ de n numere. Pentru rezolvarea problemei se poate proceda în mai multe moduri: se determină valoarea cea mai mare din prima jumătate, fie aceasta x_1 , apoi se determină valoarea cea mai mare din a doua jumătate a secvenței, fie aceasta x_2 . Soluția problemei este $\max(x_1, x_2)$. Problema inițială a fost descompusă în două subprobleme de același tip, dar “mai simple” deoarece lungimea fiecărei secvențe este jumătate din lungimea secvenței inițiale. Problema poate fi rezolvată pe baza unei metode care o “reduce” succesiv la o problemă “mai simplă”, determinându-se valoarea maximă din primele $n-1$ componente ale secvenței (fie aceasta x_1), valoarea maximă fiind $\max(x_1, a_n)$. Ambele soluții sunt recursive și sunt reprezentate prin funcțiile Pascal *max1* și *max2*.

```
function max1(var a:vector; s,d:byte):real;
var x1, x2:real;
begin
  if s=d then max1:=a[s]
  else
    begin
      x1:=max1(a,s,(s+d)div 2);
      x2:=max1(a,(s+d)div 2+1,d);
      if x1>x2 then max1:=x1
      else max1:=x2;
    end;
end;

function max2(var a:vector; n:byte):real;
var x1:real;
begin
  if n=1 then max2:=a[1]
  else
    begin
      x1:=max1(a,n-1);
      if x1>a[n] then max1:=x1
      else max1:=a[n];
    end;
end;
```

Un alt exemplu în care, pentru rezolvarea problemei, se poate efectua un raționament similar este următorul. Se presupune că ecuația $f(x)=0$ are o singură soluție x_0 în intervalul (a,b) . Se dorește obținerea unei valori aproximative \hat{x} astfel încât $|x_0 - \hat{x}| < \varepsilon$, pentru $\varepsilon > 0$ dat. Deoarece ecuația $f(x)=0$ are o singură soluție x_0 în intervalul $[a,b]$, rezultă că $f(a)f(b)<0$ și, de asemenea, dacă pentru $a<\alpha<\beta<b$ $f(\alpha)f(\beta)<0$, atunci $x_0 \in (\alpha,\beta)$. Pe baza acestei proprietăți, dacă c este mijlocul

intervalului (a,b) , atunci este îndeplinită una și numai una dintre relațiile $f(a)f(c)<0$, $f(c)=0$, $f(b)f(c)<0$. Dacă $f(c)=0$, atunci $x_0=c$. Dacă $f(a)f(c)<0$ atunci $x_0 \in (a,c)$, altfel $x_0 \in (c,b)$. Se presupune că $f(a)f(c)<0$. Se poate aplica același procedeu intervalului (a,c) și se continuă până când sau este obținută soluția exactă sau intervalul care conține soluția este de lungime inferioară lui ε . În cazul în care terminarea calculului se realizează prin identificarea unui interval de lungime inferioară lui ε , \hat{x} poate fi considerat oricare dintre numerele din acel interval, de exemplu mijlocul intervalului. Evident, numărul maxim de iterații N pentru obținerea preciziei ε rezultă din inegalitatea $\frac{b-a}{2^N} < \varepsilon$, adică $N = \left\lceil \log_2 \left(\frac{b-a}{\varepsilon} \right) \right\rceil + 1$.

Se presupune că funcția f este calculată prin apelul funcției Pascal $f(a:real):real$. O variantă recursivă a metodei descrise este:

```
uses crt;
{$F+}

type
  fct=function(x:real):real;

var
  eps,a,b,x:real;
  functie:fct;

function f(x:real):real;
begin
  f:=x*x*x-8;
end;

procedure bisectie(a,b,eps:real;var f:fct;var x:real);
var xx:real;
begin
  if f(a)=0 then x:=a
  else if f(b)=0 then x:=b
  else if b-a<eps then x:=(a+b)/2
  else
    begin
      xx:=(a+b)/2;
      if f(xx)*f(a)<0 then bisectie(a,xx,eps,f,x)
      else bisectie(xx,b,eps,f,x)
    end;
  end;

begin
  clrscr;
  write('Introduceti a ');
  readln(a);
  write('Introduceti b ');
  readln(b);
  eps:=exp(-20);
  functie:=f;
  bisectie(a,b,eps,functie,x);
  writeln('Solutia este:',x:5:2);
  readln;
end.
```


De asemenea, rezolvarea problemei turnurilor din Hanoi a fost realizată tot pe baza unei metode de reducere și anume: problema deplasării a n discuri $P(n)$ a fost redusă succesiv la rezolvarea problemelor mai simple $P(n-1)$, $P(n-2)$, ..., $P(1)$.

Unul dintre cei mai eficienți algoritmi pentru sortarea crescătoare a unei secvențe de numere reale este cunoscut sub numele de algoritmul de *quicksort* (*sortare rapidă*) și reprezintă, de asemenea, un exemplu de aplicare a metodei *divide et impera*. Fie secvența $(v_p, v_{p+1}, \dots, v_u)$, unde inițial $p=1$, $u=n$ (n =dimensiunea vectorului).

Dacă $p = u$, secvența este sortată.

Altfel, se poziționează v_p în această secvență astfel încât toate elementele ce ajung în fața lui să fie mai mici decât el și toate cele care îi urmează să fie mai mari decât el; fie poz poziția lui corectă în secvența $(v_p, v_{p+1}, \dots, v_{poz}, v_{poz+1}, \dots, v_n)$. Procedeu se reia pentru secvențele $(v_p, v_{p+1}, \dots, v_{poz-1})$ și $(v_{poz+1}, v_{poz+2}, \dots, v_n)$, deci $p \leq poz-1$ și $poz+1 \leq u$.

Poziționarea elementului v_p se face astfel: se utilizează doi indicatori, i și j ; inițial $i = p$ și $j = u$. Se compară v_i cu v_j , dacă nu este necesară interschimbarea, se micșorează j cu 1, repetându-se procesul; dacă apare o interschimbare, se mărește i cu 1 și se continuă compararea, mărin i până la apariția unei noi interschimbări. Apoi se micșorează din nou j , continuându-se în același mod până când $i = j$.

Exemplu:

5.1. Aplicarea algoritmului de sortare rapidă pentru

$v = (10, 12, 19, 15, 3, 17, 4, 18)$

1 2 3 4 5 6 7 8

$p = 1, u = n = 8$

a) poziționarea lui $v_p = v_1 = 10$

$i=1, j=8$

Sensul de parcurgere: \leftarrow

$i=1, j=8 \ 10 < 18 \Rightarrow j=j-1$

$i=1, j=7 \ 10 > 4 \Rightarrow$ se efectuează interschimbarea și se atribuie $i=i+1=2$

$v=(4, 12, 19, 15, 3, 17, 10, 18)$

$i=2, j=7$

Sensul de parcurgere: \rightarrow

$i=2, j=7 \ 12 > 10 \Rightarrow$ se efectuează interschimbarea și se atribuie $j=j-1=6$

$v=(4, 10, 19, 15, 3, 17, 12, 18)$

$i=2, j=6$

Sensul de parcurgere: \leftarrow

$i=2, j=6 \ 10 < 17 \Rightarrow j=j-1$

$i=2, j=5 \ 10 > 3 \Rightarrow$ se efectuează interschimbarea și se atribuie $i=i+1=3$

$v=(4, 3, 19, 15, 10, 17, 12, 18)$

$i=3, j=5$

Sensul de parcurgere: \rightarrow

$i=3, j=5 \ 19 > 10 \Rightarrow$ se efectuează interschimbarea și se atribuie $j=j-1=4$

$v=(4, 3, 10, 15, 19, 17, 12, 18)$

$i=3, j=4$

Sensul de parcurgere: \leftarrow

$i=3, j=4 \quad 10 < 15 \Rightarrow j=j-1$

$i=3, j=3 \quad i=j$, poziționare realizată

b) Se obține vectorul:

$v=(4, 3, 10, 15, 19, 17, 12, 18)$

1 2 3 4 5 6 7 8

$poz=3$, se lucrează cu secvențele

(4, 3) și (15, 19, 17, 12, 18)

Primul nivel de recursie:

Pentru $Sv1=(4, 3)$

$p=1, u=2$

a1) poziționarea lui 4:

$i=1, j=2$

Sensul de parcurgere: \leftarrow

$i=1, j=2 \quad 4 > 3 \Rightarrow$ se efectuează interschimbarea și se atribuie $i=i+1=2$

$sv1(3, 4)$, $poz=2$

$i=j=2$, stop

b1) se ajunge la secvențele (3) și \emptyset ; acestea sunt sortate

Pentru $sv2 = (15, 19, 17, 12, 18)$

4 5 6 7 8

a1) poziționarea lui 15

$i=4, j=8$

Sensul de parcurgere: \leftarrow

$i=4, j=8 \quad 15 < 18 \Rightarrow j=j-1=7$

$i=4, j=7 \quad 15 > 12 \Rightarrow$ se efectuează interschimbarea și se atribuie $i=i+1=5$

$sv2=(12, 19, 17, 15, 18)$

4 5 6 7 8

$i=5, j=7$

Sensul de parcurgere: \rightarrow

$i=5, j=7$

$19 > 15 \Rightarrow$ se efectuează interschimbarea și se atribuie $j=j-1=6$

$sv2=(12, 15, 17, 19, 18)$

4 5 6 7 8

$i=5, j=6$

Sensul de parcurgere: \leftarrow

$i=5, j=6 \quad 15 < 17 \Rightarrow j=j-1$

$i=5, j=5 \quad i=j$ poziționare realizată

$sv2=(12, 15, 17, 19, 18)$

4 5 6 7 8

$poz = 5$

b1) Se obține secvența $sv2=(12, 15, 17, 19, 18)$

4 5 6 7 8

$poz=5$; se lucrează mai departe cu secvențele

sv3=(12) sortată

sv4=(17, 19, 18)
 6 7 8

În acest moment, v=(3, 4, 10, 12, 15, 17, 19, 18)

Al doilea nivel de recursie - avem numai secvența

sv4=(17, 19, 18)
 6 7 8

a2) poziționarea lui 17

i=6, j=8

Sensul de parcurgere: ←

i=6, j=8 17<18 ⇒ j=j-1=7

i=6, j=7 19<19 ⇒ j=j-1=6

i=j=6 - poziționare realizată

b2) Se obține sv4=(17, 19, 18), poz=6,
 6 7 8

se lucrează mai departe cu secvențele

sv5= sortată

sv6=(19, 18)
 7 8

În acest moment v=(3, 4, 10, 12, 15, 17, 19, 18)

Al treilea nivel de recursie - avem numai secvența

sv6=(19, 18)
 7 8

a3) poziționarea lui 19

i=7, j=8

Sensul de parcurgere: ←

i=7, j=8 19>18 ⇒ se interschimbă și i=i+1=8

sv6=(18,19)

i=j=8, poziționare realizată

b3) S-a obținut secvența sv6=(18, 19), poz=8

Următoarele secvențe sunt sv7=(18), sv8= ∅ , ambele sortate

Calculul se încheie, v=(3, 4, 10, 12, 15, 17, 18, 19) este sortat crescător.

```
program quick_sort;  
  uses crt;
```

```
  var
```

```
    x:array[1..100] of integer;  
    n,i:byte;
```

```
  procedure poz(p,u:byte; var k:byte);
```

```
  var
```

```
    i,j:byte;  
    l,di,dj:shortint;  
    {di, dj: pasii de incrementare pentru i si j;  
      ei indica sensul parcurgerii}  
    v:integer;
```

```

begin
  i:=p; j:=u; di:=0; dj:=-1;
  while i<j do
    if x[i]>x[j] then
      begin
        v:=x[i];
        x[i]:=x[j];
        x[j]:=v;
        l:=di; di:=-dj; dj:=-l;
        i:=i+di; j:=j+dj;
      end
    else begin
      i:=i+di; j:=j+dj;
    end;
  end;
  k:=i;
end;

procedure quick(p,u:byte);
var
  i:byte;
begin
  if p>u then
  else begin
    poz(p,u,i);
    quick(p,i-1);
    quick(i+1,u);
  end;
end;

begin{ program principal}
clrscr;
write('Dimensiunea vectorului:');
readln(n);
for i:=1 to n do
  read(x[i]);
quick(1,n);
for i:=1 to n do
  write(x[i], ' ');
end.

```

În programul *quick_sort*, pentru simularea sensurilor de parcurgere, s-a lucrat cu incremenți pentru i și j , desemnați prin d_i , respectiv d_j . Pentru sensul de parcurgere “←” (de la sfârșitul secvenței spre începutul ei) i rămâne constant ($d_i=0$) și j este decrementat cu o unitate ($d_j=-1$). Pentru celălalt sens, j rămâne constant ($d_j=0$) și este incrementat i ($d_i=1$). Trecerea de la un sens la celălalt, efectuată în momentul unei interschimbări, determină secvența de operații:

```

l=dj;
di=dj;
dj=-l;

```

adică d_i este interschimbabil cu $-d_j$. Vectorul de sortat x este variabilă globală procedurii *quick*.

Soluțiile recursive propuse în exemplele precedente se bazează fie pe o metodă de tip reducere, fie pe o metodă de descompunere. În toate cazurile a fost posibilă “sinteza” unei soluții a problemei date din soluțiile subproblemelor la care problema

s-a redus, respectiv în care s-a descompus. De asemenea, pentru fiecare dintre problemele considerate au fost definite subproblemele primitive (condițiilor terminale) a căror soluție este “cunoscută” sau dată. Metoda de rezolvare se numește *divide et impera* (dezbină și stăpânește) și semnifică ideea prin care este realizată construcția soluției.

5.3 Metoda backtracking

Pentru rezolvarea anumitor probleme este necesară desfășurarea unui proces de căutare a soluției aflate într-o anumită mulțime, numită *spațiul stărilor*. Pentru fiecare element din spațiul stărilor este definită o mulțime de acțiuni sau alternative. Momentul inițial în rezolvarea problemei corespunde unei stări, numită *inițială*, iar soluțiile corespund drumurilor în spațiul stărilor, de la cea inițială până la una finală. Procesul de rezolvare a problemei poate fi imaginat ca o secvență de acțiuni care asigură “deplasarea” (prin intermediul unei secvențe de stări) în spațiul stărilor din starea inițială la cea finală. În cazul anumitor probleme se dorește obținerea unei singure soluții, altele solicită determinarea tuturor soluțiilor sau determinarea unei soluții optime, dintr-un anumit punct de vedere (soluție optimală).

Se presupune că problema constă în a ajunge în vârful unui munte pornind de la baza lui. În general, există mai multe puncte din care se desprind mai multe poteci, dar nu neapărat toate conduc spre vârful muntelui. În acest caz, starea inițială este baza muntelui, există o singură stare finală și anume vârful muntelui, spațiul stărilor incluzând și toate punctele de ramificare a drumului. O soluție poate fi apreciată ca optimală din mai multe puncte de vedere: un drum care solicită cel mai mic efort din partea celui care-l urmează; un cel mai scurt drum; un drum care trece printr-un punct preferat etc.

Un alt exemplu este un labirint având una sau mai multe ieșiri. Starea inițială poate fi considerată orice cameră a labirintului, problema revenind la găsirea unui drum din camera respectivă către una dintre ieșiri. Desfășurarea procesului de căutare a unei stări finale presupune, la fiecare etapă, alegerea opțiunii pentru o alternativă posibilă a stării curente și detectarea acelor stări “capcană” din care nu mai este posibilă continuarea procesului, sau deja se cunoaște excluderea atingerii unei stări finale. Detectarea stării “capcană” trebuie să determine revenirea la starea din care s-a ajuns la ea și selectarea unei noi opțiuni de continuare. În cazul în care nu mai există alternative care să nu fi fost selectate anterior, o astfel de stare devine la rândul ei “capcană” și pentru ea se aplică același tratament.

Prin soluție a problemei se înțelege o secvență de acțiuni care determină tranziția din starea inițială într-o stare finală, fiecare componentă a unui *drum soluție* reprezentând o alternativă din mulțimea de variante posibile. Cu alte cuvinte, x_1 este alternativa aleasă pentru starea inițială, x_2 este alternativa selectată pentru starea în

care s-a ajuns pe baza opțiunii x_l ș.a.m.d. După efectuarea acțiunii corespunzătoare alegerii alternativei x_n rezultă o stare finală.

Forma standard a metodei corespunde unei probleme în care trebuie găsit un drum soluție $x=(x_1, x_2, \dots, x_n)$ cu $x_i \in S_i$, unde fiecare mulțime S_i este finită și conține s_i elemente. În plus, se presupune că fiecare S_i este ordonată și reprezintă mulțimea alternativelor existente la momentul i al căutării.

În anumite cazuri interesează obținerea unei singure soluții, în altele sunt căutate toate soluțiile problemei sau cele care îndeplinesc un criteriu dat (de exemplu, se maximizează sau minimizează o funcție f definită pe mulțimea drumurilor soluție din spațiul stărilor).

Procesul de căutare a unui drum soluție revine la tentativa de extindere a porțiunii de drum construit, alegând prima alternativă disponibilă pentru starea curentă atinsă. Continuarea drumului poate fi realizată până la atingerea unei stări finale sau până la întâlnirea unei stări *capcană* (mulțimea vidă de alternative). Dacă este atinsă o stare *capcană*, atunci este necesară revenirea la starea anterioară și selectarea următoarei alternative disponibile acestei stări. Dacă nu mai există alternative disponibile, atunci se inițiază o nouă revenire ș.a.m.d. În cazul în care există cel puțin încă o alternativă disponibilă, atunci se reia procesul de extindere a drumului rezultat. În condițiile în care revenirea poate conduce la atingerea stării inițiale și pentru ea nu mai există alternative disponibile, se consideră că problema nu are soluție.

Pentru implementarea căutării este necesară reținerea alternativei selectate pentru fiecare stare atinsă până la cea curentă, astfel încât, în cazul unei reveniri să fie posibilă alegerea alternativei următoare. Cu alte cuvinte, procesul de căutare revine la tentativa de extindere a drumului curent (pasul de continuare), cu eventuala revenire în cazul atingerii unei stări *capcană* (pasul de revenire - *back*), memorând alternativele selectate pentru fiecare stare intermediară atinsă (*track*). De aici își are geneza numele metodei *backtracking*.

Pentru determinarea unei singure soluții, descrierea pe pași a metodei este:

- *starea inițială* a problemei este prima alternativă posibilă pentru starea curentă ; fie aceasta $x_1 \in S_1$;
- dacă starea curentă rezultată prin alternativa x_1 este finală, atunci $x=(x_1)$ este soluție; stop;
- altfel, este selectată prima alternativă din mulțimea de acțiuni posibile pentru starea curentă; fie aceasta $x_2 \in S_2$;
- dacă secvența de alternative care a condus la starea curentă este $x=(x_1, x_2, \dots, x_k)$, atunci:
 - dacă starea curentă este finală, soluția este $x=(x_1, x_2, \dots, x_k)$; stop;
 - altfel
 - B1: dacă pentru starea curentă există alternative disponibile, atunci se alege prima dintre ele și se continuă;

- B2: altfel, se revine la starea anterioară celei curente, soluția parțial construită devine $x=(x_1, x_2, \dots, x_{k-1})$ și se efectuează B1.
- dacă, în urma unui pas de revenire, s-a ajuns la starea inițială și nu mai sunt alternative disponibile, atunci problema nu are soluție; stop.

În cazul în care trebuie determinate toate soluțiile problemei, căutarea continuă după determinarea fiecărei soluții prin efectuarea de reveniri succesive. Terminarea căutării este decisă în momentul în care s-a revenit la starea inițială și nu mai există alternative disponibile.

Dacă se dorește obținerea numai a soluțiilor care optimizează o funcție criteriu f , atunci metoda se aplică pentru determinarea tuturor soluțiilor problemei, fiecare nouă soluție rezultată fiind comparată cu “cea mai bună” soluție determinată anterior. Pentru aceasta este necesară reținerea “cele mai bune” soluții calculate la fiecare moment.

Forma generală a metodei backtracking este implementată de procedura *back*.

```
procedure back(k:byte);
begin
  if k=n+1 then final
  else
    begin
      x[k]:=init(k);
      while succ(k) do
        if continuare(k) then back(k+1);
      end;
    end;
end;
```

în care:

- *final* este o procedură care descrie prelucrarea dorită pentru o soluție determinată (se afișează rezultatul, se testează o funcție criteriu pentru soluția obținută *samd*);
- *init(k)* efectuează inițializarea lui x_k cu o valoare prin care se indică faptul că, până la acel moment, nu a fost selectată nici o alternativă pentru poziția k ;
- *succ(k)* este o funcție booleană care calculează *true*, dacă și numai dacă există succesor pentru x_k în S_k ;
- *continuation(k)* este o funcție booleană pentru testarea condițiilor de continuare; calculează *true* dacă și numai dacă este posibilă extinderea drumului curent.

În continuare sunt prezentate câteva probleme rezolvate prin metoda backtracking.

1. Să se genereze toate permutările mulțimii $\{1, 2, \dots, n\}$.

În acest caz, $S_1 = S_2 = \dots = S_n = \{1, 2, \dots, n\}$. Alternativele posibile pentru starea inițială corespund alegerilor pentru prima poziție dintr-un vector soluție. Pentru fiecare k , $1 \leq k \leq n-1$, dacă $x=(x_1, x_2, \dots, x_k)$ este drumul calculat până la

momentul k , atunci $x_i \neq x_j, \forall 1 \leq i \neq j \leq k$ și alternativele posibile pentru starea curentă sunt elementele x_{k+1} din mulțimea $\{1, 2, \dots, n\}$ care îndeplinesc cerința $x_i \neq x_{k+1}, \forall 1 \leq i \leq k$.

De exemplu, pentru $n=3$, soluțiile problemei sunt:

$x_1=(1,2,3)$, $x_2=(1,3,2)$, $x_3=(2,1,3)$, $x_4=(2,3,1)$, $x_5=(3,1,2)$, $x_6=(3,2,1)$.

Funcția *init(k)* realizează inițializarea elementului $x[k]$ cu valoarea 0, pentru a marca faptul că, până la momentul curent, nu a fost selectată nici o alternativă pentru $x[k]$. Funcția *succ(k)* calculează *true* dacă elementul $x[k]$ are succesori în mulțimea $\{1, 2, \dots, n\}$, caz în care acesta este determinat prin incrementare. Altfel, funcția calculează *false*. Funcția *continuare(k)* returnează *true* dacă și numai dacă secvența (x_1, x_2, \dots, x_k) calculată până la momentul curent este corectă, conform regulilor descrise anterior.

Conform schemei generale, programul Pascal este:

```
program permutare;
uses crt;
type tip_elem=0..7;
var x:array[1..7] of tip_elem;
    n:byte;

function init:byte;
begin
    init:=0;
end;

function succ(k:byte):boolean;
begin
    succ:=x[k]<n; {atribuire de valoare logica}
    inc(x[k]);
end;

function continuare(k:byte):boolean;
var i:byte;
begin
    i:=1;
    while(i<k)and(x[i]<>x[k]) do inc(i);
    continuare:=i=k; {atribuire de valoare logica}
end;

procedure final;
var i:byte;
begin
    for i:=1 to n do write(x[i], ' ');
    readln;
end;

procedure back(k:byte);
begin
    if k=n+1 then final
    else
        begin
            x[k]:=init;
            while succ(k) do
                if continuare(k) then back(k+1);
        end;
end;
```



```
        end;
    end;

    begin
    clrscr;
    write('Numarul de elemente ale permutarii: ');
    readln(n);
    back(1);
    end.
```

În cazul acestei probleme este posibilă operarea unor simplificări în scrierea codului, pe baza observațiilor:

- funcția *init(k)* nu depinde de valoarea lui *k* și returnează întotdeauna valoarea 0;

- funcția *succ(k)* nu depinde de valoarea parametrului *k* și realizează întotdeauna o incrementare ($S_1 = S_2 = \dots = S_n = \{1, 2, \dots, n\}$).

Procedura *back(k)* poate fi descrisă fără a utiliza funcțiile *init* și *succ*, astfel:

```
procedure back(k:byte);
var i:byte;
begin
    if k=n+1 then final
    else
        for i:=1 to n do
            begin
                x[k]:=i;
                if continuare(k) then back(k+1);
            end;
        end;
    end;
```

Înlocuirea în procedura *back* a structurii repetitive *while* cu ciclul *for* este posibilă datorită faptului că funcția *succ* realiza incrementarea valorii elementului $x[k]$.

Varianța de program Pascal rezultată în urma acestor simplificări este:

```
program permutare_1;
uses crt;
type tip_elem=0..7;
var x:array[1..7] of tip_elem;
    n:byte;

procedure final;
var i:byte;
begin
    for i:=1 to n do write(x[i], ' ');
    readln;
end;

function continuare(k:byte):boolean;
var i:byte;
begin
    i:=1;
    while (i<k) and (x[i]<>x[k]) do inc(i);
    continuare:=i=k;
end;

procedure back(k:byte);
```

```

var i:byte;
begin
  if k=n+1 then final
  else
    for i:=1 to n do
      begin
        x[k]:=i;
        if continuare(k) then back(k+1);
      end;
    end;

  begin
    clrscr;
    write('Numarul de elemente ale permutarii: ');
    readln(n);
    back(1);
  end.

```

2. Se presupune că se dispune de n tipuri de bancnote, $n \leq 20$, cu valori diferite; din fiecare tip se dispune de un număr cunoscut de bancnote. Considerându-se dată o sumă de bani s , să se determine o modalitatea de plată a sa utilizându-se un număr minim de bancnote.

Valorile și numărul de bancnote disponibile din fiecare tip sunt memorate într-o tabelă y cu 2 linii și n coloane, astfel: pentru fiecare $1 \leq i \leq n$, $y[1,i]$ reprezintă valoarea unei bancnote de tipul i și $y[2,i]$ este numărul de bancnote disponibile din tipul i . Vectorul xb memorează modalitatea optimă de plată a sumei s , adică $xb[i]$ reprezintă numărul de bancnote alese din tipul i într-o descompunere optimă; vectorul x memorează descompunerea curentă. Numărul minim de bancnote utilizate în plata sumei s este nrb .

Metoda utilizată în rezolvarea problemei este backtracking, urmărindu-se minimizarea funcției $f(x) = \sum_{i=1}^n x[i]$, în condițiile în care $\sum_{i=1}^n x[i]y[1,i] = s$ și $0 \leq x[i] \leq y[2,i]$, $i = 1, \dots, n$. Sunt generate toate descompunerile posibile ale sumei s funcție de bancnotele disponibile și, la fiecare moment în care este determinată o astfel de descompunere, aceasta este comparată cu precedenta. În vectorul xb este memorată o cea mai bună descompunere pe baza criteriului f , după fiecare astfel de comparație.

```

program bancnote;
uses crt;

var x,xb:array[1..100] of longint;
    y:array[1..2,1..20] of longint;
    n,i,nrb:word;
    s:longint;

function init(k:word):longint;
begin
  init:=-1;
end;

function urm(k:word):boolean;
begin

```

```
        urm:=x[k]<y[2,k];inc(x[k]);
    end;

    function continuare(k:word):boolean;
    var
        sc,i:longint;
    begin
        sc:=0;
        for i:=1 to k do sc:=sc+x[i]*y[1,i];
        if k<n then continuare:=sc<=s
        else continuare:=sc=s;
        end;
        procedure final(k:word);
        var i:byte;
            nr:longint;
        begin
            nr:=0;
            for i:=1 to n do
                begin
                    nr:=nr+x[i];
                end;
            if nr<nrb then
                begin
                    for i:=1 to n do xb[i]:=x[i];
                    nrb:=nr;
                end;
            end;
        end;

    procedure back(k:byte);
    begin
        if k=n+1 then final(k)
        else
            begin
                x[k]:=init(k);
                while urm(k) do
                    if continuare(k) then back(k+1)
                end;
            end;
        end;

    begin
        clrscr;
        write('Dati numarul de bancnote');
        readln(n);
        writeln('Valorile si numarul de exemplare:');
        for i:=1 to n do
            begin
                write('Valoarea:');readln(y[1,i]);
                write('Numar de bancnote:');readln(y[2,i]);
            end;
        write('Suma schimbata:');
        readln(s);
        nrb:=maxint;
        back(1);
        writeln('Numarul minim de bancnote:',nrb);
        writeln;
        writeln('Numarul de bancnote alese din fiecare tip:');
        for i:=1 to n do
            writeln('Tip ',i,' numar de bancnote alese:',xb[i]);
        end.
```