ALGORITMICĂ

Lector Dr. Paul Iacob

Asistent Drd. Lucian Sasu

Universitatea Transilvania din Brasov
Facultatea de Matematică şi Informatică
Catedra de Informatică

Octombrie 2003

Cuprins

Capitolul 1

Introducere

1.1 Scopul cursului

Lucrarea de faţă îşi propune să fie un nurs introductiv în teoria algoritmilor. Se vor introduce noţiunile de algoritm, subalgoritm, complexitatea algoritmilor, diferite metode de elaborare a algoritmilor, probleme clasice. Cursul este detinat studenţilor din anul întâi, atât de la zi cât şi de la forme alternative (învăţământ la distanţă, cursuri postuniversitare).

1.2 Generalităţi

Pentru rezolvarea unei probleme pe calculator, se parcurg următorii paşi:

1. se detectează problema;
2. se adaugă restricţiile impuse de domeniul care a creat problema;
3. se creează un model matematic;
4. se concepe un algoritm (un mod) de rezolvare a problemei;
5. se scrie un program;
6. se interpreateză rezultatele de la calculator în termeni specifici domeniului problemei.

Paşii 1, 2, 3 sunt în general reuniţi sub numele de *analiză*. Paşii 4, 5 constituie *programarea*. Pasul 6 face parte din *implementare*.

Deci algoritmul reprezintă un pas intermediar între modelul matematic şi programul realizat într-un limbaj conceput pentru calculator. Originea

cuvântului "algoritm" este legată de numele Abu Ja'far Mohammed Ibn Mûsâ al Khowârizmî, autor arab al unei cărţi de matematică. Noţiunea de algoritm este fundamentală, deci nu se poate defini; în general, ea este explicată ca o succesiune finită de paşi computaţionali care transformă date de intrare în date de ieşire.

Proprietăţile pe care trebuie să le aibe un algoritm sunt:

1. Fiecare pas trebuie să conţină o operaţie bine definită (de exemplu următoarele operaţii nu sunt bine definite: "4/0" sau "scade 10 sau 20 din 30");

2. Pasul care urmează fiecărui pas trebuie să fie unic definit;

3. Fiecare pas trebuie să fie efectuabil, adică să poată fi executat cu creionul pe hârtie: operaţii cu numere întregi - da, dar operaţii cu numere reale în general - nu.

4. Trebuie să se termine în timp finit. Se poate spune că timpul trebuie să fie rezonabil de scurt şi totuşi vor fi consideraţi algoritmi care pe maşinile actuale necesită zeci de ani pentru că se presupune că viitoarele maşini vor fi mult mai rapide.

Ce vom studia în legătură cu algoritmii?

1. Cum să concepem un algoritm? Crearea unui algoritm nu va fi niciodată pe deplin automatizată, va rămâne o problemă a omului în primul rând şi abia apoi a maşinii. În această ordine de idei vom studia în acest curs "metode de elaborare a algoritmilor".

2. Exprimarea noastră va fi restrânsă, din necesitatea de a scrie uşor programe corecte, la algoritmi structuraţi, adică algoritmi care folosesc anumite structuri specifice care vor fi expuse la începutul cursului.

3. Vom învăţa să verificăm corectitudinea algoritmilor concepuţi. Un algoritm corect trebuie să rezolve problema propusă pentru orice date de intrare. Nu se pune problema verificării cu anumite date (acest lucru ţine de verificarea programului), ci problema unei demonstraţii a corectitudinii programului (în general ca o teoremă).

4. Nu ajunge să construim algoritmi corecţi! Trebuie să cunoaştem şi performanţele algoritmilor creaţi. Pot exista mai mulţi algoritmi care să rezolve aceeaşi problemă, unii mai greu sau mai uşor de înteles (de urmărit), unii mai lungi în numărul de paşi descrişi, dar cu performanţă urmărită prin *analiză* (numărul de paşi elementari executaţi, în legătură

cu dimensiunea intrării). O altă problemă este cea legată de dimensiunea memoriei necesare execuţiei programului realizat după un anumit algoritm.

Capitolul 2

Limbajul psedocod şi structuri fundamentale

2.1 Obiective

În acest capitol se va prezenta lexicul limbajului pseudocod, care va fi folosit în restul lucrării pentru reprezentarea algoritmilor. Se introduc instrucţiunile de atribuire, decizie, ciclare, citire, scriere. Noţiunile prezentate sunt însoţite de exemple simple care să ajute la însuşirea temeinică a elementelor.

2.2 Limbajul pseudocod

Limbajul în care este descris algoritmul trebuie să fie un intermediar între limbajul matematic şi cel de programare.

Limbajul *schemei logice* are avantaje de explicaţie grafică, dar şi dezavantaje legate de structurare şi translatare, aşa că vom folosi *pseudocodul*. Acest limbaj foloseşte cuvinte cheie (subliniate), cuvinte care definesc variabile şi constante şi text explicativ (între acolade).

Putem distinge următoarea structură generală:

Start Numele programului
  citeşte(date de intrare)
  prelucrează datele
  scrie(rezultate)
Stop

Putem specifica:

Liste de date	(a, b, x)
Vectori	$(a(i), i = 1, n)$
Matrici	$((x(i, j), j = 1, n), i = 1, m)$
Constante	('Viteza=', v)

Operaţia cea mai des folosită este *atribuirea*.

$$a = 1$$

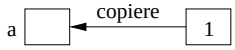


Figura 2.1: Atribuirea $a = 1$.

În acest mod, numele a este legat de o zonă de memorie în care se introduce valoarea 1. Pentru

$$a = b + c$$

efectul este dat în figura 2.2:

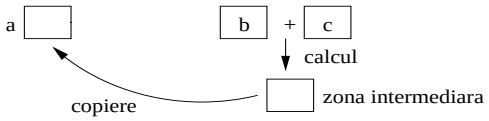


Figura 2.2: Atribuirea $a = b + c$.

Se calculează rezultatul operaţiei de adunare $b + c$, acesta depunându-se într-o zonă intermediară. Rezultatul adunării se copiază apoi în zona de memorie alocată variabilei a. Chiar şi pentru o atribuire de genul:

$$a = a + 1$$

rezultatul adunării lui a cu 1 se depune într-o zonă intermediară, după care se copiază în locaţia lui a, analog cu figura 2.2.

Atunci când pasul următor depinde de o anumită situaţie cucerită până atunci, el este rezultatul unei decizii care va fi scris astfel:

dacă condiţie logică *atunci*
(grup de paşi 1)
altfel
(grup de paşi 2)
sfârşit dacă

Exemplul 1.

Start Algoritm 1
  citeşte(i, a)
  dacă $i \leq 1$ *atunci*
    $a = a + i$
  altfel
    $a = a - i$
  sfârşit dacă
  scrie('a=', a)
Stop

Acest program scrie $\mathbf{a = 10}$ dacă intrarea este $0, 10$, iar pentru intrarea $2, 10$ scrie $\mathbf{a = 8}$.

Exemplul 2. Următorul algoritm găseşte cel mai mare dintre trei numere:

Start Algoritm 2
  citeşte(a, b, c)
  $max = a$
  dacă $max < b$ *atunci*
    $max = b$
  sfârşit dacă
  dacă $max < c$ *atunci*
    $max = c$
  sfârşit dacă
  scrie('Maximul este ', max)
Stop

Urmăriţi algoritmul pentru valorile de intrare $(10, 1, 3)$, $(2, 2, 5)$, $(1, 12, 2)$.

Există grupuri de instrucţiuni care se execută în mod repetat. Aceste structuri se numesc cicluri şi apar des în elaborarea algoritmilor.

Prima formă de ciclare se numeşte ciclu cu numărator şi se foloseşte atunci când se ştie de câte ori se execută un grup de instrucţiuni. Are forma:

pentru $ind = vi, vf, pas$
  grup de paşi
sfârşit pentru

Dacă $pas = 1$, atunci el se poate omite. Grupul de paşi din interiorul ciclului poate să nu se execute niciodată, dacă $vi > vf$ (pentru $pas > 0$), respectiv dacă $vi < vf$ (pentru $pas < 0$). De exemplu, ciclul de mai jos nu se execută dacă $n < 1$:

pentru $ind = 1, n$
grup de paşi
sfârşit pentru

Exemplul 3. Algoritmul de mai jos calculează suma componentelor unui vector $a(i), i = 1, n$:

Start Algoritm 3
citeşte($n, (a(i), i = 1, n)$)
$s = 0$ {0 este element neutru pentru adunare}
pentru $i = 1, n, 1$
$s = s + a(i)$
sfârşit pentru
scrie('suma este ', s)
Stop

Exemplul 4. Pentru a număra componentele pozitive (np), negative (nn) şi cele nule (nz) ale unui vector vom avea următorul algoritm:

Start Algoritm 4
citeşte($n, (a(i), i = 1, n)$)
$np = 0$
$nn = 0$
$nz = 0$
pentru $i = 1, n$
dacă $a(i) > 0$ *atunci*
$np = np + 1$
altfel
dacă $a(i) < 0$ *atunci*
$nn = nn + 1$
altfel
$nz = nz + 1$
sfârşit dacă
sfârşit dacă
sfârşit pentru
scrie('Pozitive= ', np, 'Negative= ', np, 'Nule= ', nz)
Stop

Acest tip de cicluri se numesc cicluri cu numărător. Mai avem şi cicluri cu condiţie, care poate fi înainte sau după corpul ciclului (grupul de paşi ce se execută de fiecare dată).

Ciclul cu test anterior are forma:

cât timp (condiţie logică)

grup de paşi
sfârşit *cât timp*

Corpul ciclului poate să nu se execute niciodată, dacă codiţia logică este de la început falsă.

Exemplul 5. Algoritmul următor (algoritmul lui Euclid) serveşte la determinarea celui mai mare divizor comun a două numere:

Start Euclid
  citeşte(a, b)
  $m = a$
  $n = b$
  cât timp $n \neq 0$
    $r = rest(m/n)$
    m=n
    n=r
  sfârşit *cât timp*
  scrie('CMMDC dintre ', a, ' şi ' , b, ' este ', m)
Stop

Urmăriţi algoritmul pentru intrările $(10, 3)$ şi $(258, 12)$.

Ciclul cu test posterior are forma:

repetă
  grup de paşi
până când condiţie logică

Grupul de paşi se execută cel puţin o dată.

Exemplul 6. Se cere listarea combinărilor din $\{1, 2, 3, 4, 5\}$ luate câte 3:

Start Combinări
  pentru $i = 1, 3$
    $j = i$
    repetă
      $j = j + 1$
      $k = j$
      repetă
        $k = k + 1$
        scrie(i,'-',j,'-',k){afişează o combinaţie}
      până când $k = 5$
    până când $j = 4$
  sfârşit *pentru*
Stop

2.3 Rezumat

Un algoritm are o structură bine definită. Instrucţiunile prezentate, împreună cu noţiunea de subalgoritm din capitolul 3 formează un vocabular suficient pentru scrierea algoritmilor. S–au prezentat: forma generală a unui algoritm, citirea şi scrierea datelor, atribuirea, instrucţiunea de decizie, cele trei forme de ciclare.

Capitolul 3

Analiza algoritmilor. Subalgoritmi. Notaţia asimptotică

3.1 Obiective

Acest capitol prezintă modalitatea de comparare a performanţei algoritmilor, folosind noţiunea de complexitate. Se introduc cele trei notaţii asimptotice împreună cu relaţiile dintre ele, formule de calcul general, proprietăţi. Exemplificarea se face prin compararea a doi algoritmi care rezolvă aceeaşi problemă: sortarea unui vector de numere (problemă de larg interes, care nu are o rezolvare trivială).

Se introduce de asemenea noţiunea de subalgoritm, absolut necesară în scrierea algoritmilor recursivi sau ca formă de abstractizare. Capitolul se încheie cu o suită de probleme propuse spre rezolvare, a căror soluţionare este dată la sfârşitul lucrării.

3.2 Analiza algoritmilor

Putem uşor să ne dăm seama că pentru a rezolva o anumită problemă pot exista mai mulţi algoritmi. De exemplu, se ştie deja că pentru aflarea celui mai mare divizor comun a două numere se poate folosi fie algoritmul lui Euclid, fie un algoritm bazat pe descompunerea numerelelor în produse de factori primi.

Se pune întrebarea: care din aceşti algoritmi este mai bun? şi în general: cum putem compara doi algoritmi care rezolvă aceeaşi problemă?

Cu răspunsul la această întrebare se ocupă analiza algoritmilor.

A analiza un algoritm înseamnă a preciza ce resurse (de memorie şi timp) o să aibă nevoie atunci când va fi translatat în program care rezolvă pe calculator problema respectivă.

Vom parcurge treptele spre noţiunile de bază ale analizei algoritmilor studiind doi algoritmi de sortare. Primul de care ne ocupăm este sortarea prin inserţie.

Pentru problema sortării avem: *la intrare* un şir de n valori numerice $(a_1, a_2, \ldots, a_n)$ şi vrem *la ieşire* un şir $(a'_1, a'_2, \ldots, a'_n)$, obţinut prin permutarea valorilor iniţiale, astfel încât $a'_1 \leq a'_2 \leq \cdots \leq a'_n$. Ideile algoritmului descris mai jos (sortare prin interclasare) pot fi înţelese dacă ne gândim la aranjarea cărţilor de joc: avem pe masă, cu faţa în jos, o serie de cărţi pe care le putem lua una câte una în mână, deplasându-le până la locul pe care trebuie să îl ocupe.

Exemplu: Fie şirul $A =$ | 3 | 1 | 5 | 2 | 7 | 4 |.Acest şir va fi parcurs de un cursor i (se ia câte o carte) şi mutat în cadrul şirului cu ajutorul unui cursor j care parcurge pe A' până la locul potrivit:

- pentru $i = 1$:
  $A' =$ | **3** | | | | | |
- pentru $i = 2$, elementul $A(2) = 1$ este mai mic decât 3, deci trebuie pus în faţa lui:
  $A' =$ | **1** | 3 | | | | |
- pentru $i = 3$:
  $A' =$ | 1 | 3 | **5** | | | |
- pentru $i = 4$:
  $A' =$ | 1 | **2** | 3 | 5 | | |
- pentru $i = 5$:
  $A' =$ | 1 | 2 | 3 | 5 | **7** | |
- pentru $i = 6$:
  $A' =$ | 1 | 2 | 3 | **4** | 5 | 7 |

Să observăm că nu avem nevoie de un spaţiu suplimentar pentru rezultat şi putem face ceea ce se numeşte sortare "in place" (pe loc) (i.e. în afara şirului nu se ţin decât cel mult un număr constant de elemente ale şirului). În pseudocod algoritmul poate fi scris astfel:

```
   Start Sortins
1  citeşte( n, (A(i), i = 1, n) )
```

```
2     pentru i = 2, n
3           k = A(i)
            {inserează A(i) în şirul sortat A(1), ..., A(i − 1)}
4           j = i − 1
5           cât timp j > 0 şi A(j) > k
6                 A(j + 1) = A(j)
7                 j = j − 1
            sfârşit cât timp
8           A(j + 1) = k
            {a terminat inserarea}
      sfârşit pentru
9     scrie( A(i), i = 1, n )
      Stop
```

Dintre resursele consumate ne va interesa în primul rând *timpul de rulare*. Timpul de rulare al unui algoritm pentru o anumită intrare (instanţă) este exprimat prin numărul de paşi elementari executaţi.

Pentru început vom considera ca pas elementar fiecare linie parcursă din algoritmul în pseudocod şi timpul pentru un astfel de pas o constantă diferită pentru fiecare linie (deşi este evident că unele din aceste constante sunt egale).

Începem prin a prezenta algoritmul *Sortins* cu timpul fiecărei instrucţiuni şi numărul de execuţii ale liniei respective. Vom nota cu t_i numărul de execuţii ale ciclului de la linia 5 (evident că el este diferit pentru fiecare valoare a lui i).

Urmărind tabelul de mai jos, putem calcula timpul total de execuţie:

Număr linie	Instrucţiune	Timp	Numar de operaţii
	Start Sortins		
1	*citeşte* ($n, (A(i), i = 1, n)$)	c_1	$n+1$
2	*pentru* $i = 2, n$	c_2	$n-1$
3	$k = A(i)$	c_3	$n-1$
4	$j = i-1$	c_4	$n-1$
5	*cât timp* $j > 0$ şi $A(j) > k$	c_5	$\sum_{i=2}^{n} t_i$
6	$A(j+1) = A(j)$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
	sfârşit cât timp		
8	$A(j+1) = k$	c_8	$n-1$
	sfârşit pentru		
9	*scrie* ($A(i), i = 1, n$)	c_9	n
	Stop		

$$\begin{aligned} T(n) &= c_1 \cdot (n+1) + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + \\ &+ c_5 \cdot \sum_{i=2}^{n} t_i + (c_6 + c_7) \cdot \sum_{i=2}^{n} (t_i - 1) + (c_6 + c_7) \cdot \sum_{i=2}^{n} (t_i - 1) + \\ &+ c_8 \cdot (n-1) + c_9 \cdot n \end{aligned}$$

Cum putem să-l aflăm pe t_i? O se ne punem, de fapt, problema intervalului în care se "plimbă", studiind cele două cazuri care corespund limitelor intervalului.
Cel mai rapid caz se obţine când şirul este gata sortat şi deci în linia 5 vom avea $A(j) \leq k$ pentru orice $j = 2, \ldots, n$ deci în acest caz $t_i = 1$, iar

$$T(n) = a \cdot n + b, \text{ unde } a, b \text{ constante, deci } T \text{ funcţie liniară de } n.$$

Cel mai "rău" caz se obţine când şirul dat este sortat descrescător, ceea ce înseamnă că pe linia 5 vom avea $A(j) > k$ pentru orice j de la $i-1$ la 1, deci $t_i = i - 1$; atunci $\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$ şi $\sum_{i=2}^{n} (i-2) = \frac{(n-2)(n-1)}{2}$, deci

$$T(n) = a \cdot n^2 + b \cdot n + c, \text{ funcţie pătratică de } n.$$

Analiza se face, în general, *pentru cel mai "rău" caz*.

Să vedem acum dacă sortarea se poate face altfel, eventual mai repede. Pentru aceasta vom adopta o altă metodă, "divide et impera". Această

metodă porneşte de la ideea că problema pusă pentru o intrare mare (număr mare de date) este mai greu de rezolvat decât o problemă cu un număr redus de date, de exemplu un şir format dintr-un singur element. Principiul "divide et impera" funcţionează astfel:

- se împarte problema în mai multe subprobleme

- se rezolvă fiecare din subprobleme (în general recursiv)

- se combină soluţiile paţiale într-o soluţie finală

Vom reveni cu amănunte despre această metodă în capitolul 8.

Algoritmul *sortării prin interclasare* (mergesort) se bazează pe metoda "divide et impera". Să vedem ce înseamnă interclasarea: din două şiruri sortate crescător să se creeze şirul sortat tot crescător care conţine toate elementele celor două şiruri iniţiale.

Aceasta dă următorul algoritm:

Start Interclasare
  citeşte($n, (A(i), i = 1, n)$)
  citeşte($m, (B(j), j = 1, m)$)
  $k = 0$
  $i = 1$
  $j = 1$
  repetă
    $k = k + 1$
    dacă $A(i) < B(j)$ *atunci*
      $C(k) = A(i)$
      $i = i + 1$
    altfel
      $C(k) = B(j)$
      $j = j + 1$
    sfârşit dacă
  până când $i > n$ sau $j > m$
  {Ciclul de mai jos poate să nu se execute nici o dată}
  pentru $il = i, n$
    $k = k + 1$
    $C(k) = A(il)$
  sfârşit pentru
  {Ciclul de mai jos poate să nu se execute nici o dată}
  pentru $jl = j, m$
    $k = k + 1$
    $C(k) = B(jl)$

sfârşit pentru
{Din cele două cicluri *pentru* de mai sus, se execută exact unul singur}
scrie($C(kl), kl = 1, m + n$)
Stop

Se observă că timpul total de execuţie pentru algoritmul de interclasare este $T(n, m) = c \cdot (n + m)$ unde c este timpul maxim necesar execuţiei instrucţiunilor din interiorul ciclurilor *repetă* şi *pentru* (c nu depinde de m, n).

Pentru algoritmul de sortare prin interclasare prezentăm noţiunea de subalgoritm.

3.3 Algoritmi parametrizaţi (subalgoritmi)

Rareori un algoritm, ca aceia prezentaţi până acum, constituie probleme independente; în general, algoritmii pentru rezolvarea unor probleme complexe au părţi care se combină între ele.

Pentru punerea în evidenţă a unei astfel de structuri modulare vom folosi algoritmi parametrizaţi numiţi şi subalgoritmi.

Parametrii vor fi variabile ale căror valori vin din algoritmul care cheamă şi se întorc înapoi în acesta, deci efectele produse asupra parametrilor de intrare vor exista şi după ce subalgoritmul se termină[1].

Algoritmul *Interclas* parametrizat ar fi: Interclas(A, n, B, m, C) ceea ce înseamnă: "interclasează A (având n componente) cu B (având m componente) şi obţine C".

Într-un algoritm parametrizat nu mai au ce căuta instrucţiunile de intrare şi nici instrucţiunea *Stop*; în locul acesteia din urmă vom folosi *Return* care înseamnă "întoarce-te în (sub)algoritmul chemător".

Exemplu: Interclas(A', n', B', m', C').

Corespondenţa dintre parametrii de chemare şi cei din subalgoritm se face poziţional.

Pentru uşurarea exprimării, vom considera şi subalgoritmi de tip funcţie care returnează o valoare. În acest caz, valoarea care se determină de către subalgoritm este atribuită numelui funcţiei.

Exemplu:

ArieCerc(r)
$ArieCerc = \pi * r * r$
Return

Apelul unui astfel de subalgoritm se face la fel ca mai înainte. În plus, valoarea returnată poate fi folosită într–o expresie sau într–o atribuire:

[1]Relativ la limbaje de programare: transmiterea parametrilor se face prin adresă.

$STotal = ArieCerc(10) * 5$

În continuare vom folosi o variantă specială a interclasării numită *Merge(A, p, q, r)*. Acest algoritm interclasează elementele $A(p), \ldots, A(q)$ cu elementele $A(q+1), \ldots, A(r)$ şi pune rezultatul în locul $A(p), \ldots, A(r)$.

Aşa cum am precizat la începutul acestui capitol algoritmul sortării prin interclasare *Mergesort* se bazează pe metoda "divide et impera", ai cărei paşi au fost deja prezentaţi. Evidenţiem în continuare paşii acestui algoritm:

1. Împarte şirul iniţial în două şiruri
2. Rezolvă (recursiv) fiecare din cele două subprobleme
3. Combină (interclasează) cele două subşiruri sortate pentru a obţine şirul sortat

Exemplu: fie şirul: | 2 | 1 | 5 | 3 |

1. împărţire: | 2 | 1 | | 5 | 3 |
2. sortare (recursiv)
   (a) împăţire: | 2 | | 1 |
   (b) Sortarea este evidentă, fiecare parte având câte un element
   (c) interclasare | 1 | 2 |

   (a) împăţire: | 5 | | 3 |
   (b) Sortarea este evidentă, fiecare parte având câte un element
   (c) interclasare | 3 | 5 |
3. interclasare: | 1 | 2 | 3 | 5 |

Algoritmul va fi:

Mergesort(A, p, r)
dacă $p < r$ *atunci*
  $q = \left[\frac{p+r}{2}\right]$ {partea întreagă}
  {q este indicele elementului de la jumătatea şirului}
  cheamă Mergesort(A, p, q)
  cheamă Mergesort$(A, q+1, r)$
  cheamă Merge(A, p, q, r)
sfârşit dacă
Return

Acest algoritm poate fi folosit de exemplu astfel:

Start Sortare
citeşte($n, (A(i), i = 1, n)$)
cheamă Mergesort$(A, 1, n)$
scrie('Şirul sortat este: ', $(A(i), i = 1, n)$)
Stop

Observaţii:

1. Sortarea constă de fapt din interclasarea perechilor de şiruri de un element, de două elemente, ..., până când interclasăm o pereche de şiruri de $\frac{n}{2}$ elemente, obţinând şirul sortat.

2. Dacă la un moment dat un şir are un număr impar de elemente, atunci se va împărţi în două subşiruri de dimeniuni care diferă printr-o unitate (subşirurile sunt cât mai echilibrate ca număr de elemente).

Verificaţi execuţia algoritmului pentru şirul

3	2	5	6	4	1	0	7

3.4 Notaţia asimptotică

Pentru a vedea cum se comportă timpul de execuţie al unui algoritm pentru intrări mari vom defini funcţiile Θ, Ω, O.

Spunem că $T(n) = \Theta(f(n))$ dacă $\exists N, c_1, c_2 > 0$ astfel încât:

$$0 < c_1 f(n) < T(n) < c_2 f(n),\ \forall n > N \tag{3.1}$$

Spunem că $T(n) = \Omega(f(n))$ dacă $\exists N, c_1 > 0$ astfel încât:

$$0 < c_1 f(n) < T(n),\ \forall n > N \tag{3.2}$$

Spunem că $T(n) = O(f(n))$ dacă $\exists N, c_1 > 0$ astfel încât:

$$T(n) < c_1 f(n),\ \forall n > N \tag{3.3}$$

Lăsăm în seama cititorului să demonstreze proprietăţile de mai jos.

Fie $F \in \{\Theta, \Omega, O\}$. Atunci:

1. Tranzitivitate:

$$T(n) = F(f(n)) \text{ şi } f(n) = F(g(n)) \Rightarrow T(n) = F(g(n))$$

2. Reflexivitate:

$$T(n) = F(T(n))$$

3. Simetrie:
$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

4. Bisimetrie:
$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$
$$f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$$

5. Dacă
$$\exists M \text{ astfel încât } g(n) > f(n),\ \forall n > M$$
atunci
$$\Theta(f(n) + g(n)) = \Theta(g(n))$$

6. $T(n) = \Theta(f(n)) \Leftrightarrow \{T(n) = O(f(n))$ şi $T(n) = \Omega(f(n))\}$

7. $\Theta(cf(n)) = \Theta(f(n)), c$ constantă > 0

Aplicând proprietăţile 1-7 se demonstrează imediat următoarele:

1. $T(n) = a \cdot n^2 + b \cdot n + c \Rightarrow T(n) = \Theta(n^2)$

2. $T(n) = c_1 \cdot n \cdot \log n + c_2 \cdot n + c_3 \Rightarrow T(n) = \Theta(n \cdot \log n)$

Din cele de mai sus, precum şi analiza făcută la algoritmul de sortare prin inserţie rezulta că complexitatea acestuia este $T(n) = \Theta(n^2)$.

Calculul complexităţii algoritmului de sortare prin interclasare se efectuează mai jos.

Când un algoritm conţine o apelare a lui însuşi se numeşte recursiv. În acest caz timpul de execuţie se descrie printr-o ecuaţie recursivă. Pentru rezolvarea unei asemenea ecuaţii vom folosi metode matematice care vor fi descrise în capitolul dedicat metodei "Divide et impera".

Ne vom limita acum la a scrie recurenţa pentru sortare prin interclasare. Dacă timpul pentru un şir cu n elemente este $T(n)$, atunci timpul pentru şirul cu $\frac{n}{2}$ elemente va fi $T\left(\frac{n}{2}\right)$, deci:

$$T(n) = \begin{cases} a & \text{, dacă } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n & \text{, dacă } n > 1 \end{cases}$$

În capitolul 8 vom demonstra că $T(n) = \Theta(n \log_2 n)$. Deci complexitatea algoritmului Mergesort este $\Theta(n \log_2 n)$.

În final, deducem că complexitatea algoritmului de sortare prin inserţie ($\Theta(n^2)$) este mai mare decât cea a algoritmului Mergesort ($\Theta(n \log_2 n)$), altfel spus *Mergesort* este mai performant decât sortarea prin inserţie.

3.5 Rezumat

Pentru compararea a doi algoritmi, cel mai utilizat criteriu este cel al timpului de execuţie relativ la dimensiunea datelor de intrare. Calculul complexităţii unui algoritm este o etapă obligatorie a soluţiei acestuia. Pentru exprimarea complexităţii se folosesc notaţiile $O(\cdot), \Omega(\cdot), \Theta(\cdot)$ care sunt legate de viteza de variaţie a timpului cerut pentru rezolvarea unei instanţe, în funcţie de dimensiunea intrării şi care sunt cu atât mai corecte cu cât n este mai mare.

Subalgoritmii sunt o modalitate de a concepe blocuri de cod parametrizate, apelabile prin intermediul unei interfeţe (liste de parametri de apel). Ele efectuează prelucrări asupra datelor transmise, rezultatul returnat fiind folosit mai apoi de algoritmul apelant. De asemenea, reprezintă o formă de abstractizare, utilizată în toate paradigmele de programare.

3.6 Exerciţii (rezolvările la pagina 135)

1. Determinaţi complexitatea algoritmilor din capitolul 2.
2. Se defineşte produsul scalar a doi vectori A şi B, de aceeaşi dimensiune n cu formula $PS = \sum_{i=1}^{n} A(i) \cdot B(i)$. Să se scrie algoritmul pentru calculul produsului scalar.
3. Se dă n, vectorul $A(i), i = 1, n$ şi un număr x. Să se scrie un algoritm care găseşte indicele j pentru care $A(j) = x$ (problema căutării).
4. Cum se modifică algoritmul de mai sus dacă ştim că vectorul este sortat crescător?
5. Se dau n şi vectorul $A(i), i = 1, n$. Să se găsească cel mai mic element.
6. Reluaţi problema anterioară în situaţia în care trebuie să aflaţi câte elemente minime sunt.Încercaţi să rezolvaţi problema folosind o singură parcurgere a şirului.
7. Se dau n şi vectorul $A(i), i = 1, n$. Ce număr apare cel mai des şi de câte ori?
8. Cum se modifică algoritmul de mai sus dacă vectorul este sortat crescător?
9. Se dau n, $((A(i, j), j = 1, n), i = 1, n)$ (A este o matrice pătratică, cu n linii şi n coloane). Să se calculeze produsul elementelor diferite de 0:

(a) din toată matricea;

(b) de pe diagonala principală;

(c) de sub diagonala principală.

Capitolul 4

Structuri elementare de date

4.1 Obiective

Am folosit, până acum, structuri de date foarte simple cum ar fi:

- variabile simple (dintre care unele aveau domeniul de valori format doar din două valori)
- vectori cu componente sortate sau nesortate (aşa cum sunt folosiţi în matematică)
- matrici considerate ca vectori de vectori sau ca masive biortogonale

Vom studia acum alte structuri de date necesare în algoritmii ulteriori: lista, stiva, coada, arborii binari, împreună cu diferite forme de reprezentare a acestora.

4.2 Structuri liniare

O structură liniară este o mulţime de $n \geq 0$ componente $x(1), x(2), \ldots x(n)$ cu proprietăţile:

1. când $n = 0$ spunem că structura este vidă;
2. dacă $n > 0$ atunci $x(1)$ este primul element iar $x(n)$ este ultimul element;
3. oricare ar fi $x(k)$ unde $k \in \{2, \ldots, n-1\}$ există un predecesor $x(k-1)$ şi un succesor $x(k+1)$.

Ne va interesa să executăm cu aceste structuri următoarele operaţii:

- adăugarea unui element;

- extragerea unui element;

- accesarea unui element $x(k)$ din structură;

- combinarea a două sau mai multe structuri într-una singură;

- "ruperea" unei structuri în mai multe structuri;

- sortarea elementelor unei structuri;

- căutarea unui element al structurii care are anumite proprietăţi;

- operaţii specifice

4.3 Stiva

Una din cele mai cunoscute structuri liniare este stiva. O stivă este caracterizată prin *disciplina de intrare şi de ieşire*. Să considerăm o mulţime de cărţi puse una peste alta; există o primă carte care se poate lua foarte uşor (TOP) şi o carte carte care se poate lua numai dacă se înlătură toate celelate cărţi (BOTTOM).

Disciplina unei stive este "ultimul intrat, primul ieşit" (disciplina care nu v-ar place să fie respectată când staţi la rând la lapte!), prescurtat LIFO (Last In, First Out).

Diferenţele faţă de un vector sunt:

- un vector are o lungime fixă, nonzero, cunoscută aprioric;

- o stivă poate fi vidă;

- stiva are un număr variabil de elemente în timpul execuţiei unui algoritm

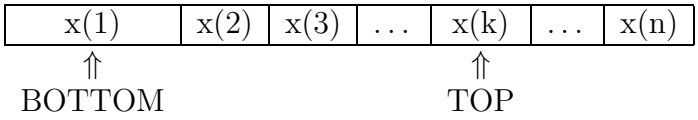
Se pune problema reprezentării concrete a unei stive în memoria unui calculator. Putem aloca o stivă în două moduri:

1. Secvenţial

2. Înlănţuit

4.3.1 Alocarea secvenţială a stivei

Folosim vectorul ca fiind structura cea mai apropiată de structura reală a memoriei. În vectorul $(x(i), i = 1, n)$ considerăm:



adică din cele n componente ale unui vector doar primele k elemente fac parte din stivă.

Algoritmul de intrare în stivă este:

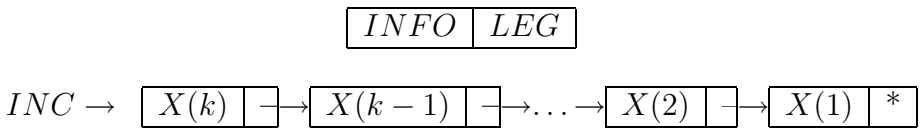
$\underline{a \Rightarrow \text{Stiva}}$
dacă $k = n$ *atunci*
  "Depăşire"
altfel
  $k = k + 1$
  $x(k) = a$
sfârşit dacă
Return

Algoritmul de ieşire (scoatere) din stivă este:

$\underline{a \Leftarrow \text{Stiva}}$
dacă $k = 0$ *atunci*
  "Stiva vidă"
altfel
  $a = x(k)$
  $k = k - 1$
sfârşit dacă
Return

4.3.2 Alocarea înlănţuită a stivei

În alocarea înlănţuită fiecare element al structurii este însoţit de adresa de memorie la care se află precedentul element. Vom folosi semnul * cu sensul "nici o adresă de memorie". Vom avea vârful stivei pus în evidenţă (ancorat) în INC şi elementul de la baza stivei va avea conţine în câmpul LEG *, ca în figură:



(ordinea $1, 2, \ldots k$ este ordinea intrării în stivă).

În acest caz intrarea în stivă va folosi stiva de locuri libere[1] (această stivă se numeşte $LIBERE$), pentru a obţine noi locuri la introducerea în stivă.

Vom prezenta în continuare algoritmii de intrare/ieşire dintr-o stivă în cazul în care alocarea ei a fost înlănţuită.

Algoritmul de intrare în stivă este:

$\underline{a \Rightarrow \text{Stiva}}$
{a este informaţia introdusă în stivcă}
$y \Leftarrow LIBERE$ {alocare de memorie pentru elementul nou}
$INFO(y) = a$
$LEG(y) = INC$
$INC = y$
Return

La început, $INC =*$, stiva neconţinând nici un element.

Algoritmul de ieşire din stivă este:

$\underline{a \Leftarrow Stiva}$
{a este informaţia extrasă din vârful stivei nevide}
dacă $INC =*$ *atunci*
  "Stiva este vidă"
altfel
  $a = INFO(INC)$
  $aux = INC$
  $INC = LEG(INC)$
  $aux \Rightarrow LIBERE$ {Eliberarea spaţiului de memorie}
sfârşit dacă
Return

4.4 Coada

O altă structură de date folosită în concepţia algoritmilor este coada. O coadă este caracterizată şi ea de o disciplină de intrare/ieşire, bineînţeles diferită de cea a stivei. De data aceasta puteţi să vă gândiţi la coada la lapte: primul care s-a aşezat al coadă va fi primul servit, adică primul care iese din coadă.

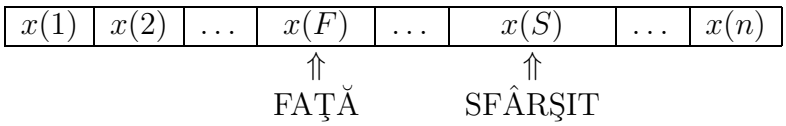
Disciplina unei cozi este deci "primul venit, primul plecat" ("first in, first out" - FIFO).

[1] Cunoscută în cadrul limbajelor de programare drept *heap*; obţinerea de memorie se face prin *alocare dinamică*, eliberarea ei prin *dealocare*.

O coadă poate fi vidă şi are şi ea un număr variabil de elemente în timpul execuţiei unui algoritm.

4.4.1 Alocarea secvenţială a cozii

Coada alocată secvenţial îşi va găsi locul tot într-un vector $(X(i), i = 1, n)$:



Din cele n componente ale vectorului, doar componentele $x(F), \ldots, x(S)$ fac parte din coadă.

Algoritmul de intrare în coadă este:

a ⇒ Coada
dacă $S = n$ *atunci*
  "Depăşire"
altfel
  $S = S + 1$
  $x(S) = a$
sfârşit dacă
Return

Algoritmul de ieşire din coadă este:

a ⇐ Coada
dacă $F > S$ *atunci*
  "Coada este vidă"
altfel
  $a = x(F)$
  $F = F + 1$
sfârşit dacă
Return

Se poate imagina uşor că procedând în acest mod (scoţând din faţă şi introducând la sfârşit), coada "migrează" spre dreapta şi poate să ajungă în situaţia de depăşire când de fapt mai există mult loc gol (în vectorul x) pe primele poziţii. Apare astfel ideea de a folosi elementele vectorului ca şi cum ar fi dispuse circular, precum în figura 4.1, pagina 34.

Lăsăm în seama cititorului să scrie algoritmii de intrare/ieşire din coada circulară.

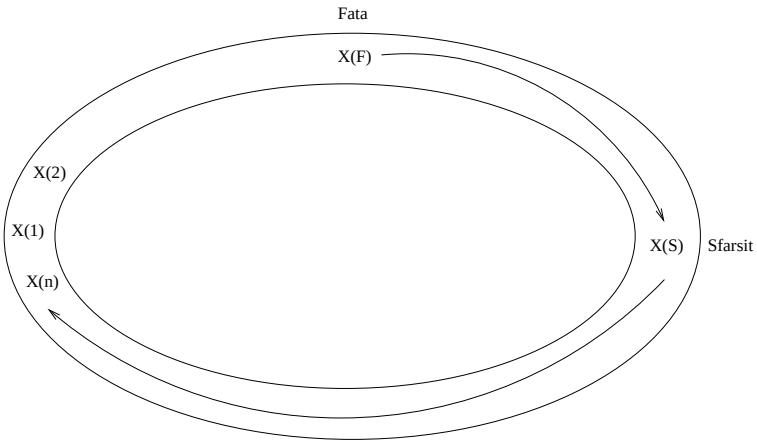
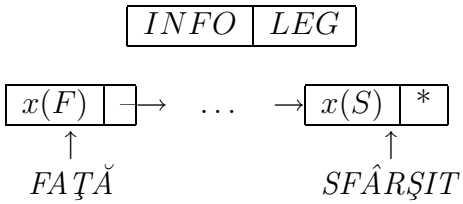


Figura 4.1: Coada circulară

4.4.2 Alocarea înlănţuită a cozii

Alocarea înlănţuită se face similar cu alocarea înlănţuită a stivei în noduri de tipul:



Algoritmul de intrare în coadă este:

a ⇒Coada
y ⇐ *LIBERE*
$INFO(y) = a$
$LEG(y) = *$
dacă FAŢĂ= atunci*
  {Coada este vidă}
  FAŢA=y
altfel
  LEG(SFÂRŞIT)=y
sfârşit dacă
SFÂRŞIT=y
Return

Iniţial, coada este vidă, i.e. *FAŢĂ=SFÂRŞIT=**.

Algoritmul de ieşire din coadă este:

a ⇒ Coada
dacă FAŢA= atunci*
  "Coada este vidă"
altfel
  $a = INFO(FAŢA)$
  $aux = FAŢA$
  $FAŢA \Rightarrow LEG(FAŢA)$
  $aux \Rightarrow LIBERE$
sfârşit dacă
Return

4.5 Arbori binari

Un arbore în care orice vârf (nod) are 0, 1 sau 2 descendenţi se numeşte *arbore binar*. Un arbore binar care are vârfuri cu 0 sau 2 descendenţi se numeşte *arbore binar strict*. Un astfel de arbore se poate folosi, de exemplu, la reprezentarea în memorie a unei expresii aritmetice.

Exemplu: expresia

$$(a+b)\cdot c - \frac{2}{c^3}$$

se va reprezenta astfel:

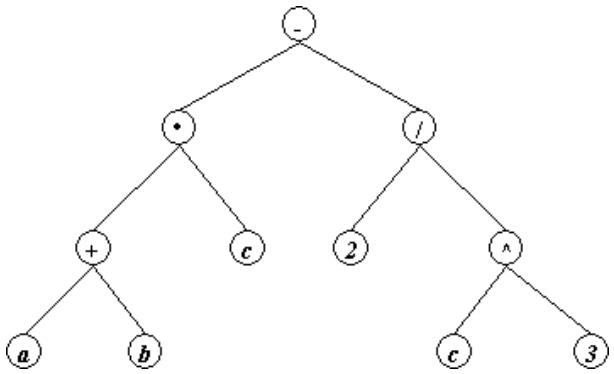


Figura 4.2: Arborele binar ataşat expresiei aritmetice $(a+b)\cdot c - \frac{2}{c^3}$. Semnul "∧" este reprezentare pentru operatorul de ridicare la putere.

Frunzele acestui arbore (nodurile fără descendenţi) conţin operanzii, iar celelalte noduri conţin operatorii. Deoarece nu toate operaţiile sunt comutative, este foarte important dacă un nod este descendent pe stânga sau pe dreapta. Rădăcina unui arbore este nodul care nu este descendentul nimănui.

Alocarea unui arbore binar se poate face:

- secvenţial

- înlănţuit

- mixt

4.5.1 Alocarea secvenţială a unui arbore binar

În vectorul $(x(i), i = 1, n)$ vom avea următoarele reguli:

- rădăcina este în $x(1)$;

- pentru fiecare nod $x(i)$, descendentul să din stânga este $x(2 \cdot i)$ iar cel din dreapta este $x(2 \cdot i + 1)$;

- dacă nu există descendent, se pune *.

Reprezentarea secvenţială a arborelui din figura 4.2 este:

-	·	/	+	c	2	∧	a	b	*	*	*	*	c	3	*	*	...	*
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	31

Observaţie: Poziţiile $x(10), \ldots x(13)$, $x(16), \ldots x(31)$ conţin *.

Vom volosi această alocare în capitolul destinat sortării pentru a construi o structură de date asemănătoare, numită HEAP[2].

4.5.2 Alocarea înlănţuită a unui arbore binar

În acest caz se folosesc noduri de forma $\boxed{INFO \mid LS \mid LD}$, unde LS şi LD conţin adresele de memorie ale nodurilor descendent stâng, respectiv descendent drept. Reprezentarea înlănţuită a arborelui din figura 4.2 este dată în figura 4.3.

[2]A nu se confunda cu heap–ul pomenit în nota de subsol de la pagina 32

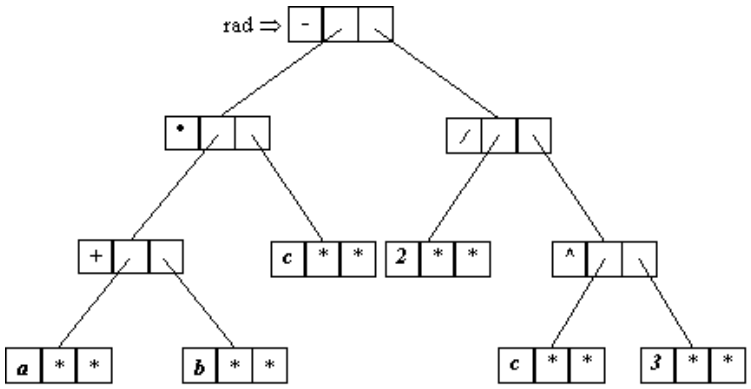


Figura 4.3: Alocarea înlănţuită pentru arborele din figura 4.2.

4.5.3 Alocarea mixtă a unui arbore binar

La acest tip de alocare se folosesc trei vectori $INFO$, LS, LD unde $LS(i)$ şi $LD(i)$ conţin indicii unde se află memoraţi descendentul stâng, respectiv descendentul drept ai nodului i.

Reprezentarea mixtă a arborelui din figura 4.2 este:

	$INFO$	LS	LD
1	-	2	3
2	$\cdot$	4	5
3	$/$	6	7
4	$+$	8	9
5	c	*	*
6	2	*	*
7	$\wedge$	10	11
8	a	*	*
9	b	*	*
10	c	*	*
11	3	*	*

La arborii binari ne interesează problema parcurgerii lor. Ţinem cont de faptul că:

- arborele binar se defineşte recursiv ca: rădăcină, subarbore stâng, subarbore drept (fiecare subarbore este la rândul lui un arbore binar, posibil vid);

- la parcurgere mai întâi va fi subarborele stâng înaintea celui drept.

După poziţia rădăcinii (R) relativ la subarborele stâng (S) şi subarborele drept (D), avem următoarele trei tipuri de parcurgeri ale unui arbore binar:

1. postordine (SRD);
2. preordine (RSD)
3. inordine (SRD)

Exemple de parcurgeri:

1. Parcurgerea arborelui din figura 4.2 în *preordine:*

$$- \quad \cdot \quad + \quad a \quad b \quad c \quad / \quad 2 \quad \wedge \quad c \quad 3$$

   Acestă scriere se numeşte *scriere poloneză prefixată* a unei expresii aritmetice.

2. Parcurgerea arborelui din figura 4.2 în *inordine:*

$$a \quad + \quad b \quad \cdot \quad c \, - \quad 2 \quad / \quad c \quad \wedge \quad 3$$

3. Parcurgerea arborelui din figura 4.2 în *postordine:*

$$a \quad b \quad + \quad c \quad \cdot \quad 2 \quad c \quad 3 \quad \wedge \quad / \quad -$$

   Acestă scriere se numeşte *scriere poloneză postfixată* a unei expresii aritmetice.

Dăm în continuare algoritmul iterativ pentru parcurgerea în preordine a unui arbore binar. Vom avea nevoie de o STIVĂ care va conţine nodurile succesive parcurse pe stânga care urmează să fie parcurse şi pe dreapta; rădacina arborelui se află în rad.

Preordine_iterativ(rad)
$STIVA = \emptyset$ {STIVA este vidă}
$nodCurent = rad$
$maiAmNoduri = adevarat$
repetă
  $parcurgPeStanga = adevarat$
  repetă
    dacă $nodCurent \neq *$ *atunci*
      scrie($INFO(nodCurent)$)
      {parcurgere pe stânga}
      $nodCurent \Rightarrow STIVA$
      $nodCurent = LS(nodcurent)$
    altfel

$parcurgPeStanga = fals$
sfârşit dacă
până când $parcurgPeStanga = fals$
{parcurgere pe dreapta}
dacă $STIVA = \emptyset$ *atunci*
$maiAmNoduri = fals$
altfel
$nodCurent \Leftarrow STIVA$
$nodCurent = LD(nodCurent)$
sfârşit dacă
până când $maiAmNoduri = adevarat$
Return

Se pot da algoritmi iterativi şi pentru celelalte două parcurgeri (vezi problema 2 de la exerciţii). De asemenea se pot da algoritmi recursivi pentru parcurgeri, mai uşor de înţeles, dar ceva mai ineficienţi (a se vedea capitolul 5, problema 6).

4.6 Rezumat

Pentru prelucrarea datelor se folosesc diferite structuri adecvate: liste, stive, cozi, arbori (sunt cele mai des utilizate, dar mai multe vor fi studiate într–un curs dedicat de structuri de date). Există diferite tipuri de reprezentări ale acestor structuri: pe bază de tablouri, folosind alocarea dinamică, sau o variantă mixtă (la arbori). Stiva şi coada au politici specifice de acces, iar arborele poate fi parcurs în moduri diferite pentru procesarea datelor.

4.7 Exerciţii (rezolvările la pagina 29)

1. Scrieţi algoritmii de intrare/ieşire din coada circulară.

2. Scrieţi algoritmii iterativi pentru parcurgerea arborilor binari în inordine şi postordine.

3. Se dă o stivă alocată înlănţuit şi o valoare x. Să se extragă din stivă, dacă există, elementul cu $INFO = x$.

4. Se dă o stivă şi o valoare x. Să se introducă în stivă după elementul cu $INFO = x$ un element cu $INFO = y$.

5. Să se implementeze o coadă folosind două stive. Să se analizeze complexitatea.

6. Să se implementeze o stivă prin două cozi. Să se analizeze complexitatea.

Capitolul 5

Recursivitate

5.1 Obiective

Capitolul detaliază modul de lucru al unui subalgoritm recursiv. Este dat un exemplu simplu de algoritm recursiv, pentru care se descriu toate informaţiile care se depun pe stivă la execuţia lui. Explicaţiile date sunt aplicabile pentru orice caz de recursivitate.

5.2 Prezentare generală

Recursivitatea reprezintă un mod de elaborare a algoritmilor, caracterizat prin faptul că un subalgoritm se apelează pe el însuşi (direct sau indirect). Ea a apărut din necesităţi practice, permiţând implementarea unei funcţii matematice recursive. Nu reprezintă o modalitate mai puternică sau mai slabă din punct de vedere al rezultatelor ce se pot obţine comparativ cu metodele iterative; s–a demonstrat că aceste două abordări sunt egale ca şi putere de calcul. Algoritmii recursivi nu sunt mai rapizi decât algoritmii iterativi echivalenţi, dar uneori sunt mai expresivi şi mai uşor de întreţinut (modificat).

Pentru înţelegerea algoritmilor recursivi, este esenţială înţelegerea lucrului cu stiva. Înainte de a se efectua un apel de subalgoritm (recursiv sau nu), se salvează pe stivă adresa de memorie a instrucţiunii de la care se va continua execuţia după întoarcerea din apel. La intrarea într–un subalgoritm se rezervă spaţiu pe stivă pentru parametrii transmişi şi pentru variabilele care au fost declarate în subalgoritm (variabile locale). La ieşirea din apel, stiva se eliberează de informaţia depusă la intrarea în apel, iar adresa următoarei instrucţiuni ce se va executa se preia de pe vârful stivei (şi de asemenea această informaţie va fi scoasă de pe stivă).

Exemplu: următorul program nu are o utilitate zdrobitoare, dar ilustrează cele spuse mai înainte. Pentru a simplifica expunerea, vom considera că transmiterea parametrilor se face prin valoare (pe stivă se copiază valorile parametrilor) şi nu prin adresă (când pe stivă se copiază adresa de memorie a parametrilor efectivi) - a se vedea convenţia de la pagina 22. În cazul exemplului de mai jos, întrucât în corpul subalgoritmului nu se modifică valoarea lui n, rezultatul ar fi acelaşi indiferent de convenţia de apel.

Start Exemplu
  citeşte(n)
  Recursiv(n)
  scrie("Sfârşit"){10}
Stop
Recursiv(n)
$i = 2 * n${i este variabilă locală, alocată în subalgoritmul *Recursiv*}
dacă $n = 0$ *atunci*
  Return
altfel
  Recursiv($n - 1$)
  $i = i + 1${100}
  scrie(i)
sfârşit dacă
Return

Ca şi comentarii sunt trecute nişte adrese la care se presupune a se afla instrucţiunile corespunzătoare. Să presupunem că se va citi $n = 2$. Evoluţia stivei programului este:

- La intrarea în algoritmul *Exemplu*, se rezervă spaţiu pe stivă pentru variabila n, în care după citire se depune valoarea 2:

$n = 2$

- Înainte de apelul *Recursiv(2)* din algoritmul *Exemplu* se scrie pe stivă adresa de memorie a instrucţiunii care se va executa după ce se va reveni din acest apel (în cazul nostru instrucţiunea de scriere cu adresa 10):

$adr = 10$
$n = 2$

- Se apelează *Recursiv(2)*; se scrie pe stivă valoara parametrului $n = 2$, se rezervă spaţiu de memorie pentru i (pentru care nu se atribuie o valoare anume la început):

i
$n = 2$
$adr = 10$
$n = 2$

Se calculează $i = 2 \cdot n = 4$, valoare care se depune în locul rezervat pentru i la acest apel (în cazul nostru, în vârful stivei):

$i = 4$
$n = 2$
$adr = 10$
$n = 2$

Se face testarea $n = 0$; deoarece este condiţia nu este îndeplinită, se intră pe ramura *altfel*.

- Avem din nou apel: înainte de acesta se salvează adresa de memorie a instrucţiunii de la care se va continua execuţia (incrementarea de la adresa 100):

$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

- Se intră în *Recursiv(1)*; se salvează pe stivă valoarea parametrului $n = 1$, se rezervă spaţiu de memorie pentru i (pentru care nu se atribuie o valoare anume la început):

i
$n = 1$
$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

Se calculează $i = 2 \cdot n = 2$, valoare care se depune în locul rezervat pentru i la acest apel (în cazul nostru, în vârful stivei):

$i = 2$
$n = 1$
$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

Se face testarea $n = 0$; deoarece este condiţia nu este îndeplinită, se intră pe ramura *altfel*.

- Avem din nou apel: înainte de acesta se salvează adresa de memorie a instrucţiunii de la care se va continua execuţia (incrementarea de la adresa 100):

$adr = 100$
$i = 2$
$n = 1$
$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

- Se intră în apelul *Recursiv(0)*; se salvează pe stivă $n = 0$, se rezervă spaţiu de memorie pentru i (pentru care nu se atribuie o valoare anume la început):

i
$n = 0$
$adr = 100$
$i = 2$
$n = 1$
$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

Se calculează $i = 2 \cdot n = 0$, valoare care se depune în locul rezervat pentru i la acest apel (în cazul nostru, în vârful stivei):

$i = 0$
$n = 0$
$adr = 100$
$i = 2$
$n = 1$
$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

Se face testarea $n = 0$; deoarece este condiţia îndeplinită, se intră pe ramura *atunci*, care reprezintă întoarcere la metoda *Recursiv(1)*.

- La întoarcerea din *Recursiv(0)* în *Recursiv(1)*, se scoate de pe stivă informaţia care a fost depusă datorită apelului *Recursiv(0)*, revenindu–se la forma de dinainte de efectuarea acestui apel:

$i = 2$
$n = 1$
$adr = 100$
$i = 4$
$n = 2$
$adr = 10$
$n = 2$

Instrucţiunea de la care se va continua execuţia este de la adresa 100, care a fost scoasă de pe stivă.

- Suntem în *Recursiv(1)*. Execuţia se continuă de la instrucţiunea cu adresa 100 (conform punctului precedent), deci i ia valoarea $2 + 1$ (valoareal 2 din sumă fiind datorată informaţiei $i = 2$ de pe stivă), care se afişează.

- Datorită instrucţiunii de retur finale din subalgoritmul *Recursiv*, avem o întoarcere de la *Recursiv(1)* la *Recursiv(2)*. Se curăţă stiva de informaţiile $i = 2$, $n = 1$, $adr = 100$ şi se continuă execuţia de la instrucţiunea cu adresa 100:

$i = 4$
$n = 2$
$adr = 10$
$n = 2$

- Suntem în *Recursiv(2)*. Execuţia se continuă de la instrucţiunea cu adresa 100, deci i ia valoarea $4+1$ (valoarea 4 din sumă fiind datorată informaţiei $i = 4$ de pe stivă), care se afişează.

- Datorită instrucţiunii de retur finale din subalgoritmul *Recursiv*, avem o întoarcere de la *Recursiv(2)* la *Exemplu*. Se curăţă stiva de informaţiile $i = 4$, $n = 2$, $adr = 10$ şi se continuarea execuţiei se va face de la instrucţiunea cu adresa 10. Stiva va fi:

  $\boxed{n = 2}$

- Se execută instrucţiunea cu adresa 10: *Scrie("Sfarşit")*.

- Se termină programul (algoritmul *Exemplu*), iar stiva devine vidă. În cazul unei implementări, controlul ar fi preluat de către sistemul de operare.

Algoritmul a afişat valorile $3, 5$. Evident, acest lucru se putea face mult mai simplu printr–o iteraţie; dar exemplul de faţă a dorit să ilustreze funcţionarea unui apel recursiv. Datorită operaţiilor de curăţare a stivei (care se efectuează printr–o simplă incrementare sau decrementare a valorii dintr–un registru al microprocesorului), precum şi datorită informaţiilor repetate care se salvează pe stivă (n, i, adr), timpul şi spaţiul de memorie necesar sunt mai mari decât pentru variantele iterative. Totuşi, în destul de multe cazuri implementările recursive sunt mult mai uşor de urmărit şi de înţeles decât cele iterative.

O observaţie importantă: programatorul este singurul răspunzător de faptul că programul lui se termină. Altfel spus, trebuie mare atenţie la impunerea unei condiţii de terminare a recursivităţii (în cazul nostru, testarea pentru $n = 0$). În caz contrar, un astfel de algoritm implementat va epuiza spaţiul disponibil pentru stivă şi se va opri fără a rezolva problema (apel recursiv la infinit).

Dacă revenim la convenţia ca transferul parametrilor să se facă prin adresă, atunci pe stivă, în cadul fiecărui subalgoritm apelat, în loc de valoarea lui n s–ar fi depus adresa de memorie rezervată în cadrul algoritmului (la noi: adresa bazei stivei). În acest fel, orice modificare care s–ar face asupra parametrului n din subalgoritmul *Recursiv* s-ar face de fapt (via această adresă pusă pe stivă) asupra valorii lui n din algoritmul *Exemplu*. Majoritatea limbajelor (Pascal, C/C++, C#, PL/SQL) au mecanisme prin care se poate stabili dacă pe stivă se depune o valoare (copia valorii parametrului efectiv) sau o adresă de memorie (adresa la care este stocat parametrul efectiv).

5.3 Exemple

5.3.1 Calcularea lui $n!$

Pentru un n natural, definim valoarea $n!$ în mod recursiv astfel:

$$n! = \begin{cases} 1, & \text{pentru } n = 0 \\ n \cdot (n-1)!, & \text{pentru } n > 0 \end{cases} \tag{5.1}$$

Algoritmul recursiv este următoarea:

Start FactRec
  citeşte(n)
  scrie($Fact(n)$)
Stop

unde subalgoritmul $Fact$ are forma:

Fact(n)
dacă $n = 0$ *atunci*
  $Fact = 1$
altfel
  $Fact = n * Fact(n-1)$
sfârşit dacă
Return

Complexitatea algoritmului este dată de ecuaţia recursivu a $T(n) = T(n-1) + c$, c fiind o constantă datorată apelului recursiv (operaţiei cu stiva) şi înmulţirii, pentru care soluţia este $T(n) = \Theta(n)$ (pentru rezolvarea recurenţei anterioare, a se vedea A). Varianta iterativă este de aceeaşi complexitate teoretică, dar mai rapidă în practică.

5.3.2 Şirul lui Fibonacci

Şirul lui Fibonacci se defineşte astfel:

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2 \end{cases} \tag{5.2}$$

Vom implementa recursiv acest algoritm:

Start FiboRec
  citeşte(n)
  scrie($Fibo(n)$)
Stop

unde algoritmul recursiv $Fibo$ este:

Fibo(n)
dacă $n = 0$ sau $n = 1$ *atunci*
  $Fibo = n$
altfel
  $Fibo = Fibo(n-1) + Fibo(n-2)$
sfârşit dacă
Return

Complexitatea teoretică este dată de ecuaţia recursivă $T(n) = T(n-1) + T(n-2) + c$, care duce la $T(n) = \Theta(\phi^n)$, unde $\phi = \frac{1+\sqrt{5}}{2}$ (tăietura de aur)(a se vedea Anexa A) . Complexitatea exponenţială mare este datorată faptului că se calculează de mai multe ori aceleaşi valori: de exemplu, în $f(n) = f(n-1) + f(n-2)$, $f(n-2)$ este apelat atât pentru calculul lui $f(n-1)$, cât şi pentru cel de al doilea membru al părţii drepte a ecuaţiei anterioare. Redundanţa de calcul devine tot mai accentuată pentru valori mici ale argumentului.

Să vedem o implementare iterativă:

Start FiboIter
  citeşte(n)
  $f(0) = 0$
  $f(1) = 1$
  pentru $i = 2, n$
    $f(i) = f(i-1) + f(i-2)$
  sfârşit pentru
  scrie($f(n)$)
Stop

Complexitatea algoritmului anterior este evident $\Theta(n)$. Acest fapt se datorează memorării valorilor lui f şi evitării calculului repetat (ca în algoritmul precedent).

5.4 Rezumat

Scrierea recursivă a algoritmilor este deseori mai uşoară decât cea iterativă. În esenţă, orice apel de subalgoritm (recursiv sau nu) determină depunerea unor informaţii pe o stivă gestionată automat. Codul rezultat este mai scurt şi mai uşor de depanat; însă o implementare recursivă poate duce la o complexitate crescută, datorită rezolvării în mod repetat a aceloraşi probleme, deci este necesară de fiecare dată o analiză pertinentă a complexităţii (un exemplu relevant este dat în cele două implementări pentru calculul şirului lui Fibonacci de mai sus).

5.5 Exerciţii (rezolvările la pagina 41)

1. Să se calculeze recursiv maximul unui şir.
2. Să se calculeze recursiv suma elementelor dintr–un şir.
3. Să se calculeze puterea a n–a a unui număr real a, cu o complexitate mai bună decât $\Theta(n)$.
4. Să se dea un contraexemplu pentru care algoritmul de la 3 nu dă numărul optim de înmulţiri.
5. Să se scrie un algoritm care calculează termenul $fibo(n)$ al şirului Fibonacci în timp mai bun de $\Theta(n)$ (a se veda şi problema 3).
6. Să se dea algoritmi recursivi pentru parcurgerea în preordine, postordine, inordine a unui arbore binar.

Capitolul 6

Metoda *Backtracking*

6.1 Obiective

Acest capitol introduce una din cele mai generale modalităţi de rezolvare a problemelor: metoda Backtracking. Ea duce la scrierea unor algoritmi care evită generarea tuturor combinaţiilor posibile şi apoi la validarea rezultatelor. Este prezentată schema generală de algoritm backtracking şi probleme clasice pentru a căror rezolvare se foloseşte această abordare.

6.2 Prezentare generală

Metoda Backtracking se poate aplica algoritmilor care trebuie să construiască o soluţie de forma

$$\mathbf{x} = (x(1), x(2), \ldots, x(n)) \tag{6.1}$$

unde pentru orice $i \in \{1, 2, \ldots, n\}$, $x(i) \in A_i$, A_i fiind o mulţime dată ca informaţie de intrare şi unde, în plus, componentele lui $\mathbf{x}$ trebuie să mai satisfacă nişte condiţii interne.

Exemplu: să considerăm:

$$\begin{aligned} A_1 &= \{A, B, E\} \\ A_2 &= \{A, C\} \\ A_3 &= \{U, S, E, C\} \end{aligned}$$

Cerinţe:

1. Să se construiască cuvintele care au prima literă din A_1, a doua literă din A_2, a treia literă din A_3;% şi care respectă condiţiile interne de la 2;

2. În plus, se cere respectarea următoarelor condiţii interne:

   (a) Nu sunt două litere consecutive egale;

   (b) Nu sunt formate numai din consoane sau numai din vocale.

Dacă luăm în considerare numai cerinţa de la 1, obţinem *soluţiile admisibile* ale problemei, de exemplu (AAU), (ACS), (EAU), (BAU), (BCC), ... Dar (AAU), (BCC) nu respectă condiţiile interne 2a, iar (EAU) nu respectă condiţia 2b.

O metodă de rezolvare a acestei probleme ar fi să construim mai întâi toate soluţiile admisibile şi apoi să verificăm care dintre ele respectă condiţiile interne. Acest mod de a aborda problema duce, în exemplul nostru, la $3 \cdot 2 \cdot 4 = 24$ de soluţii din care evident cele care încep cu AA, de exemplu, nu sunt bune.

Dacă mulţimile $A_1, A_2, \ldots, A_n$ au respectiv $k_1, k_2, \ldots, k_n$ elemente, atunci am avea în total $k_1 \cdot k_2 \ldots k_n$ soluţii admisibile. Pentru $k_1 = k_2 = \cdots = k_n = m$ avem m^n soluţii admisibile, deci un algoritm de tip *brute force* care ia în considerare toate posibilităţile ar avea o complexitate exponenţială.

Algoritmul de tip backtracking urmăreşte micşorarea numărului de operaţii, dar trebuie să spunem că rămâne de complexitate exponenţială. Acest algoritm urmăreşte să genereze câte o componentă $x(k) \in A_k$ cât timp mai are o componentă netestată din A_k; dacă nu mai are componente netestate, se revine la alegerea elementului de pe poziţia anterioară $x(k-1)$, altfel se merge la componenta $x(k+1)$. In felul acesta s-ar genera toată mulţimea soluţiilor admisibile; dar dacă după ce am găsit o componentă $x(k)$ testăm dacă bucata de vector $(x(1), x(2), \ldots, x(k))$ generată până atunci satisface *condiţiile de continuare* derivate din condiţiile interne, atunci vom putea să nu permitem generarea inutilă a unor soluţii care oricum nu ar fi bune (în exemplul nostru (AA_)).

Trebuie să mai subliniem că simplul fapt că bucata $(x(1), x(2), \ldots x(k))$ din vectorul $\mathbf{x}$ satisface condiţiile de continuare nu înseamnă că o să găsim, obligatoriu, o soluţie a problemei (există, totuşi, o metodă care găseşte soluţia pas cu pas, fără revenire, numită Greedy; ea va fi prezentată mai târziu).

Schema algoritmului general va fi:

Backtracking(A, n)
$k = 1$
cât timp $k > 0$
  dacă $k = n + 1$ *atunci*
    scrie($(x(i)), i = 1, n$){s-a generat o soluţie}
    $k = k - 1$
  altfel
    dacă mai există valori netestate $a \in A_k$ *atunci*

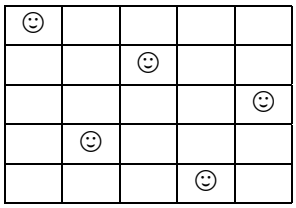
$x(k) = a$
dacă $(x(1), x(2), \ldots, x(k))$ satisface condiţiile de continuare *atunci*
$k = k + 1$
sfârşit dacă
altfel
$k = k - 1$
sfârşit dacă
sfârşit dacă
sfârşit cât timp
Return

6.3 Exemplu

Vom utiliza acest algoritm prin problema aşezării damelor pe tabla de şah astfel încât oricare două dame să nu se atace (2 dame se atacă dacă sunt pe aceeşi linie, coloană, diagonală).

Vom considera mulţimile $(A_i)_{i=1,n}$ ca fiind poziţiile (de la 1 la n) pe care le poate ocupa o damă pe linia i. Evident că de data aceasta toate mulţimile A_i sunt egale.

Deci o configuraţie pe o tablă de $n = 5$ precum mai jos va fi simbolizată de vectorul soluţie $\mathbf{x} = (1, 3, 5, 2, 4)$ şi se poate vedea că în acest caz condiţiile interne sunt respectate (oricare două dame nu se bat).



Algoritmul pentru $n > 4$ este:

Dame(n)
pentru $i = 1, n$
$x(i) = 0$ {iniţial pe nici o linie nu avem poziţionată nici o regină}
sfârşit pentru
$k = 1$
cât timp $k > 0$
dacă $k = n + 1$ *atunci*
scrie($(x(i)), i = 1, n$)
$k = k - 1$
altfel
$x(k) = x(k) + 1$

```
        dacă x(k) ≤ n atunci
          dacă DispunereCorecta(x, k)=adevarat atunci
            k = k + 1
          sfârşit dacă
        altfel
          x(k) = 0{ Nu avem nici o regină pe această linie}
          k = k − 1{ Ne întoarcem la regina de pe poziţia anterioară}
        sfârşit dacă{x(k) ≤ n}
      sfârşit dacă{ k = n + 1}
    sfârşit cât timp
    Return
```

DipunereCorecta($\mathbf{x}, k$) testează dacă condiţiile de continuare pentru bucata $x(1), \ldots, x(k)$ sunt îndeplinite; returnază adevărat dacă nu avem regine care se atacă, fals în caz contrar.

```
DispunereCorecta(x, k)
corect = adevarat{ presupunem iniţial că nu avem nici un atac între regina
k şi cele dispuse anterior}
pentru i = 1, k − 1
  dacă x(k) = x(i) sau |x(k) − x(i)| = k − i atunci
    { sunt pe aceeaşi coloană sau pe aceeaşi diagonală}
    corect = fals
  sfârşit dacă
sfârşit pentru
DispunereCorecta = corect
Return
```

6.4 Rezumat

Multe probleme cer generarea uneia/mai multora/tuturor soluţiilor care respectă anumită cerinţe. O abordare de tipul "generează toate combinaţiile, apoi reţine–le pe cele valide" (*brute–force*) este posibilă, dar şi cea mai ineficientă. Metoda backtracking urmăreşte depistarea cât mai devreme cu putinţă a soluţiilor care sunt invalide (pe măsură ce componentele lor se generează). Deşi algoritmii rezultatţi sunt de complexitate exponenţială, multe soluţii admisibile, dar invalide sunt evitate.

6.5 Exerciţii (Rezolvările la pagina 153)

1. Scrieţi un algoritm pentru parcurgerea tablei de şah cu un cal (calul sare pe diagonală unui dreptunghi cu laturile de un pătrat şi două patrate). Fiecare pătrat al tablei trebui să fie vizitat exact o singură dată.

2. Având 4 culori şi o hartă cu n ţări (dată print-o matrice de adiacenţă: $a_{ij} = 1$ dacă ţara i este vecină cu ţara j, 0 altfel), să se coloreze harta astfel ca două ţări vecine să nu aibe aceeaşi culoare (două ţări se consideră a fi vecine dacă au o frontieră comună).

3. O organizaţie are în componenţa sa n bărbaţi şi m femei. Să se scrie un algoritm care listează toate modalităţile în care se poate alcătui o delegaţie care să conţină cel puţin k femei, $k < m$.

4. Cum poate fi plătită o sumă de x lei, în bancnote de valoare $v(i)$, $i = 1, n$ din care avem câte $b(i)$ bucăţi? Să se dea toate soluţiile posibile.

Capitolul 7

Generarea submulţimilor

7.1 Obiective

De multe ori soluţiile care se cer pentru rezolvarea unei (sub)probleme sunt de o formă particulară: produse carteziene, familia submulţimilor, a combinaţiilor, aranjamentelor, permutărilor, submulţimilor de sumă dată. Acest capitol conţine rezolvări clasice pentru aceste probleme, în mai multe variante.

7.2 Schema generală de lucru

O aplicaţie imediată a metodei Backtracking este generarea submulţimilor. Vom considera sumbulţimi ale mulţimii $A = \{a(1), a(2), \ldots, a(n)\}$ cu n elemente. Evident că avem bijecţie între mulţimea indicilor $\{1, 2, \ldots, n\}$ şi elementele mulţimii A. Exemplu:

$$\{1, 3, 4, 5\} \leftrightarrow \{a(1), a(3), a(4), a(5)\}$$

Ca atare, vom presupune pentru simplificarea algoritmilor că $a(i) = i, \forall i \in \{1, \ldots, n\}$.

Avem la dispoziţie următoarele modalităţi de reprezentare a unei submulţimi X cu i elemente ale lui A:

1. Printr–un vector x cu i componente care conţine elementele submulţimii, precizându–se valoarea lui i sau completând componentele lui x cu o valoare neutilizată (de exemplu 0);

2. Analog cu metoda 1, dar plasând elemetele submulţimii la sfârşitul vectorului x;

3. Prin vectorul caracteristic al submulţimii: un vector c cu n componente unde $c(i) = 1$ arată că $a(i)$ aparţine submulţimii, iar $c(i) = 0$ arată că $a(i)$ nu aparţine submulţimii.

Exemplu: pentru $A = \{1, 2, 3, 4, 5, 6, 7\}$, submulţimea $X = \{1, 3, 4, 5\}$ poate fi reprezentată astfel:

1. $x = (1, 3, 4, 5, 0, 0, 0)$: elementele din submulţimea X la început, restul până la n poziţii fiind completate cu o valoare care nu se găseşte în A, de exemplu 0 (care nu este indice valid);

2. $x = (0, 0, 0, 1, 3, 4, 5)$: elementele din submulţimea X la sfârşit, primele poziţii fiind completate cu o valoare care nu se găseşte în A;

3. $c = (1, 0, 1, 1, 1, 0, 0)$: c este vectorul caracteristic:

$$c(i) = \begin{cases} 1, & \text{dacă } i \in X \\ 0, & \text{dacă } i \notin X \end{cases}$$

Toate generările prezentate în continuare au un caracter iterativ — fiecare nou element (*i.e.* submulţime) al mulţimii fiind găsit pe baza elementului anterior.

Algoritmul general pe care îl vom prezenta în continuare foloseşte, pe lângă un vector de dimensiune n (în care apar succesiv elementele submulţimii) şi un indicator de generare. Acest indicator de generare are iniţial valoarea 0; apoi are valoarea 1 cât timp se mai poate genera o submulţime X şi din nou valoarea 0 când s-a terminat generarea.

Generare(x, n, ig)
dacă $ig = 0$ *atunci*
  $x =$ valoarea (submulţimea) iniţială
  $ig = 1$
  Return{iese în (sub)algoritmul apelant}
sfârşit dacă
dacă există un succesor al lui x *atunci*
  $x =$succesorul lui x\{ig rămâne 1\}
altfel
  $ig = 0$\{nu s-a mai putut genera submulţimea X\}
sfârşit dacă
Return

Un mod de utilizare a acestui algoritm pentru generări este următorul:

$ig = 0$

repetă
  cheamă Generare(x, n, ig)
  dacă $ig = 0$ *atunci*
    exit{Ieşi din ciclare}
  altfel
    cheamă Prelucrare(x, n){prelucrează elementul proaspăt obţinut al submulţimii}
  sfârşit dacă
până când $fals$

În majoritatea algoritmilor pe care îi vom prezenta, vectorii vor fi generaţi în ordine lexicografică[1], ceea ce corespunde strategiei generale Backtracking.

7.3 Generarea elementelor unui produs cartezian

Fiind date mulţimile $A_1, A_2, \ldots A_m$, unde $A_i = \{1, 2, \ldots, n(i)\}$, dorim să generăm elementele produsului cartezian $A_1 \times A_2 \times \cdots \times A_m$. Vom avea în total $\prod_{i=1}^{m} n(i)$ elemente.

În algoritmul următor elementele produsului cartezian vor fi generate succesiv în vectorul x cu m elemente, unde o configuraţie $x = (x(1), x(2), \ldots, x(m))$ va desemna elementul $(A_1(x(1)), A_2(x(2)), \ldots, A_m(x(m)))$ al produsului cartezian.

Vom porni deci cu un element iniţial $x = (1, 1, \ldots, 1)$ care înseamnă că din fiecare mulţime A_i luăm primul element. În continuare vom determina cel mai mare indice i cu proprietatea că $x(i) < n(i)$ şi vom alege ca succesor al lui $x = (x(1), x(2), \ldots, x(m))$ elementul $x = (x(1), x(2), \ldots, x(i-1), x(i)+1, 1, \ldots, 1)$.

Dacă pentru fiecare $i \in \{1, \ldots, m\}$ avem $x(i) = n(i)$ înseamnă că aceasta a fost ultima componentă a produsului cartezian şi generarea s-a terminat. Algoritmul pentru generarea produsului cartezian este:

ProdusCartezian(x, m, n, ig)
dacă $ig = 0$ *atunci*
  pentru $i = 1, m$

[1]Două şiruri $x = (x(1), \ldots, x(n))$ şi $y = (y(1), \ldots, y(n))$ sunt în ordine lexicografică strict crescătoare dacă există $k \geq 1, k < n$ astfel încât $x(i) = y(i)$ pentru orice $i \leq k$ şi $x(k+1) < y(k+1)$. Exemplu: $(1, 1, 2) < (1, 2, 1)$ în sens lexicografic. Pentru cazul în care x şi y au forma generală $x = (x(1), \ldots, x(m))$ şi $y = (y(1), \ldots, y(n))$, dacă există $k \leq m, n$ astfel încât $(x(1), \ldots, x(k)) < (y(1), \ldots, y(k))$ atunci x este mai mic (lexicografic) decât y. Dacă un astfel de k nu există (adică $x(1) = y(1), \ldots, x(p) = y(p)$ unde p este $\min(m, n)$), atunci cel mai scurt dintre vectori este considerat cel mai mic. Exemplu: $(1, 2) < (1, 3, 4)$, $(1, 2) < (1, 2, 3)$.

$x(i) = 1$
sfârşit *pentru*
$ig = 1$
Return
sfârşit dacă
pentru $i = m, 1, -1$
dacă $x(i) < n(i)$ *atunci*
$x(i) = x(i) + 1$
Return
altfel
$x(i) = 1$
sfârşit dacă
sfârşit *pentru*
$ig = 0$
Return

7.4 Generarea tuturor submulţimilor unei mulţimi

O primă metodă ar fi să generăm elementele produsului cartezian al mulţimilor $A_1 = A_2 = \cdots = A_n = \{0, 1\}$. Vectorul x care se obţine, interpretat ca vector caracteristic, va fi element al mulţimii părţilor lui $\{1, 2, \ldots, n\}$

O a doua metodă ve genera submulţimile unei mulţimi în ordinea crescătoare a numărului de elemente utlizând metoda (2) de reprezentare a mulţimilor. Specificăm că vectorii x vor fi generaţi şi de această dată în ordine lexicografică strict crescătoare.

Fie $x = (x(1), x(2), \ldots, x(n))$; atunci succesorul lui se va determina în modul următor: se va determina indicele i cu proprietatea că $x(i) < i$ şi pentru $n \geq j > i$ avem $x(j) = j$. Este evident că următorul element x' în ordine lexicografică este cel în care $x'(i) = x(i) + 1$ iar celelalte componente cresc cu câte 1, adică: $x'(i + 1) = x(i) + 2$, ..., $x'(n) = x(i) + n - i + 1$.

Se începe cu $x = (0, 0, \ldots, 0)$ care reprezintă mulţimea vidă, iar atunci când $x = (1, 2, \ldots, n)$ înseamnă că am generat ultimul element.

Algoritmul pentru generarea submulţimilor unei mulţimi este:

Submultimi(x, m, ig)
dacă $ig = 0$ *atunci*
pentru $i = 1, n$
$x(i) = 0$
sfârşit *pentru*

$ig = 1$
Return
sfârşit dacă
pentru $i = n, 1, -1$
dacă $x(i) < i$ *atunci*
$x(i) = x(i) + 1$
pentru $j = i + 1, n$
$x(j) = x(j - 1) + 1$
sfârşit pentru
Return
sfârşit dacă$\{x(i) < i\}$
sfârşit pentru
$ig = 0$
Return

7.5 Generarea mulţimii combinărilor

Fie $A = \{1, 2, \ldots n\}$ şi $m \leq n$. Vom construi în continuare algoritmi de generare a tuturor celor C_n^m combinări din mulţimea A cu proprietatea că oricare două elemente diferă între ele prin natura lor, nu şi prin ordine.

O primă metodă se bazează pe metoda de reprezentare 1. Şi de această dată vectorii vor fi generaţi în ordine lexicografică începând cu $x = (1, 2, \ldots, m)$[2]. Fiind dat $x = (x(1), x(2), \ldots, x(m))$, următorul element x' va fi determinat astfel:

1. se determină indicele i cu proprietatea: $x(i) < n - m + i$, $x(i + 1) = n - m + i + 1$, $\ldots$, $x(m) = n$
2. se trece la x' prin:
   - $x'(j) = x(j)$ dacă $j < i$
   - $x'(i) = x(i) + 1$
   - $x'(j) = x(j - 1) + 1$ pentru j de la $i + 1$ la m
3. când nu găsim i cu proprietatea de mai sus înseamnă că $x = (n - m + 1, n - m + 2, \ldots, n)$ şi deci s-a generat ultimul element.

Algoritmul va fi următorul:
Combin1(x, n, m, ig)

[2]Restul componentelor vectorului x fiind permanent 0, nu vor mai fi reprezentate, deci vom lucra efectiv doar cu m componente în loc de n.

```
dacă ig = 0 atunci
   pentru i = 1, m
      x(i) = i
   sfârşit pentru
   ig = 1
   Return
sfârşit dacă
pentru i = m, 1, −1
   dacă x(i) < n − m + i atunci
      x(i) = x(i) + 1
      pentru j = i + 1, m
         x(j) = x(j − 1) + 1
      sfârşit pentru
      Return
   sfârşit dacă
sfârşit pentru
ig = 0
Return
```

A doua metodă foloseşte reprezentarea elementelor prin vectorul caracteristic, generat în ordine lexicografică crescătoare. Primul element va fi: $x = (0, 0, \ldots, 0, 1, 1, \ldots, 1)$ vectorul cu ultimele m poziţii egale cu 1 şi 0 în rest.

Fie acum vectorul x căruia vrem să îi determinăm succesorul x'; pentru aceasta vom pune în evidenţă ultima secvenţă de cifre 1:

$$X = (\ldots, \underset{a}{0, \quad 1, \quad \ldots \quad 1}, \underset{b}{\quad}, 0, \ldots, 0)$$

unde a este indicele unde începe această secvenţă iar b indicele ultimului element egal cu 1.

Trecerea la x' depinde de faptul că ultima secvenţă conţine un singur element ($a = b$) sau mai multe elemente ($a < b$).

Dacă $a = b$ atunci $x'(a-1) = 1$, $x'(a) = 0$ şi în rest $x'(i) = x(i)$ va determina configuraţia următoare.

Dacă $a < b$ atunci rezultă că $x(a-1) = 0$. Să notăm cu l_1 lungimea secvenţei de elemente 1, deci $l_1 = b - a + 1$ şi cu l_0 lungimea secvenţei de zerouri de la sfârşitul lui x, deci $l_0 = n - b$. Rezultă că $l_1 \geq 2$ şi $l_0 \geq 0$.

Trecerea de la x la x' se poate vedea uşor în următoarea schemă:

$$x = (\ldots, 0, \underset{a}{1, \quad \ldots,} \overset{l_1}{\underset{b}{1}}, 0, \overset{l_0}{\ldots, \quad 0})$$

$$x = (\ldots, 1, \underbrace{0, \ldots,}_{a} \underbrace{1,}_{n-l_1+2} \underbrace{\ldots,}_{l_1-1} 1)$$

Deci $x'(a-1)$ va deveni 1, urmează apoi un şir de l_0+1 zerouri, urmat, în final de un şir de l_1-1 elemente egale cu 1. Algoritmul pentru generarea combinărilor este următorul:

Combinari2(x, n, m, ig, a, b)
dacă $ig = 0$ *atunci*
  pentru $i = 1, n-m$
    $x(i) = 0$
  sfârşit pentru
  pentru $i = n-m+1, n$
    $x(i) = 1$
  sfârşit pentru
  $ig = 1$
  $a = n-m+1$
  $b = n$
  Return
sfârşit dacă
dacă $a = 1$ *atunci*
  $ig = 0$
  Return
sfârşit dacă
dacă $a < b$ *atunci*
  $l_1 = b-a+1$
  $x(a-1) = 1$
  pentru $i = a, n-l_1+1$
    $x(i) = 0$
  sfârşit pentru
  pentru $i = n-l_1+2, n$
    $x(i) = 1$
  sfârşit pentru
  $a = n-l_1+2$
  $b = n$
  Return
altfel
  $x(a-1) = 1$
  $x(a) = 0$
  $b = a-1$
  pentru $i = a-1, 1, -1$

$\underline{dacă}\ x(i) = 0\ \underline{atunci}$
$a = i + 1$
$\underline{Return}$
$\underline{sfârşit\ dacă}$
$a = 1$
$\underline{sfârşit}\ \underline{pentru}$
$\underline{sfârşit\ dacă}\{a < b\}$
$\underline{Return}$

7.6 Generarea mulţimii permutărilor

Complexitatea oricărui algoritm care rezolvă această problemă este de $\Omega(n!)$, deoarece trebuie generaţi $n!$ vectori. Vectorul p cu n elemente va conţine permutarea mulţimii $A = \{1, 2, \ldots, n\}$. Există mulţi algoritmi pentru rezolvarea acestei probleme, dar noi prezentăm aici numai doi dintre ei. Pentru mai multe detalii, a se consulta lucrarea [3].

Prima metodă este: urmăreşte generarea vectorului x în ordine lexicografică crescătoare. Pentru construcţia lui p' – succesorul lui p – vom pune în evidenţă ultima secvenţă descrescătoare:

$$p(i) < p(i+1) > p(i+2) > \cdots > p(n) \tag{7.1}$$

şi în consecinţă componenta care trebuie modificată este $p(i)$. În locul lui $p(i)$ trebuie pusă o componentă $p(j)$ cu $j > i$, cu condiţia ca $p(j) > p(i)$, dar pentru că p' este succesorul lui p, trebuie ales j astfel încât $p(j)$ să fie cel mai mic cu proprietatea $p(j) > p(i)$; fie acest indice k; atunci prin interschimbarea lui $p(i)$ cu $p(k)$ se ajunge la:

$$(p(1), \ldots, p(i-1), p(k), p(i+1), \ldots, p(k-1), p(i), p(k+1), \ldots, p(n))$$

unde $p(i+1) > p(i+2) > \cdots > p(k-1)$ din ecuaţia (7.1). Dacă $p(i)$ nu ar fi mai mare decât $p(k+1)$, atunci am avea (datorită inegalităţii $p(k) > p(k+1)$) că $p(k)$ nu ar fi cel mai mic din secvenţa $p(i+1), \ldots, p(n)$ care este mai mare decât $p(i)$, contradicţie cu alegerea lui k; deci avem că $p(i) > p(k+1)$, deci $p(i) > p(k+1) > \cdots > p(n)$. Uşor se demonstrează că $p(k-1) > p(i)$. Rezultă că secvenţa de la $p(i+1)$ la $p(n)$ este descrescătoare şi pentru a obţine p' nu avem decât să inversăm această secvenţă, interschimbând termenii de la început cu cei de la sfârşit până la mijlocul secvenţei. Algoritmul este:

Perm1p, n, ig
$\underline{dacă}\ ig = 0\ \underline{atunci}$
$\underline{pentru}\ i = 1, n$

$p(i) = i$
sfârşit pentru
$ig = 1$
Return
sfârşit dacă
$i = n - 1$
cât timp $p(i) > p(i+1)$
$i = i - 1$
dacă $i = 0$ *atunci*
$ig = 0$
Return
sfârşit dacă
sfârşit cât timp
$k = n$\{ căutarea lui k\}
cât timp $p(i) > p(k)$
$k = k - 1$
sfârşit cât timp
{schimbă $p(i)$ cu $p(k)$}
$p(i) \leftrightarrow p(k)$[3]
{calculează mijlocul secvenţei}
$m = \left[\frac{n-i}{2}\right]$\{$[x]$ este partea întreagă a lui x\}
pentru $j = 1, m$
$p(i+j) \leftrightarrow p(n+1-j)$
sfârşit pentru
Return

Metoda pe care o prezentăm în continuare se bazează pe generarea recursivă a permutărilor. Dacă avem S_k mulţimea permutărilor de k elemente de forma $p = (p(1), \ldots, p(k)) \in S_k$, atunci S_{k+1} va conţine pe baza fiecărei permutări p câte $k+1$ permutări P': $(p(1), \ldots, p(k), k+1)$, $(p(1), \ldots, p(k-1), k+1, p(k))$, $\ldots$, $(p(1), k+1, p(2), \ldots, p(k))$, $(k+1, p(1), \ldots, p(k))$. Deci permutările S_n vor fi frunzele unui arbore construit astfel:

- rădăcina conţine 1;
- descendenţii unei permutări $p = (p(1), \ldots, p(k))$ sunt cele $k+1$ permutări descrise mai sus

Putem ataşa acestui arbore unul în care toate permutările sunt de n elemente, prin completarea permutării de k elemente cu componentele $(k+1, k+2, \ldots, n)$.

[3]Notaţia $a \leftrightarrow b$ înseamnă "interschimbă valoarea lui a cu valoarea lui b", care se efectuează prin următoarea secvenţă: $aux = a$, $a = b$, $b = aux$.

Exemplu: pentru $n = 3$ se obţine arborele din Figura 7.1 şi arborele ataşat din Figura 7.2.

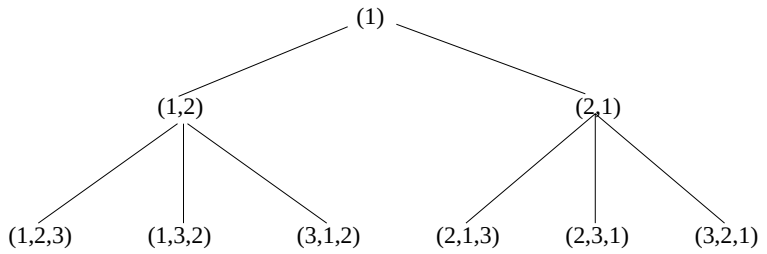


Figura 7.1: Arborele ataşat unei permutări de 3 elemente

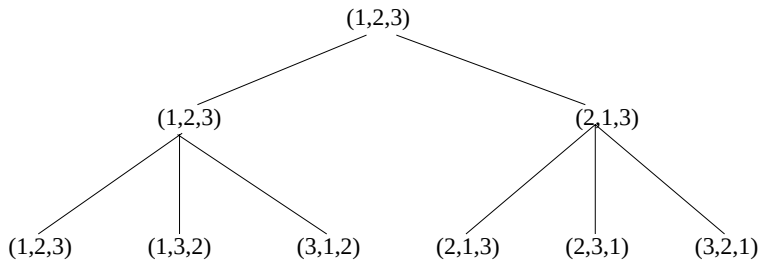


Figura 7.2: Arbore cu noduri completate până la n

Arborele ataşat are, evident, aceleaşi vârfuri terminale. Trecerea de la o permutare p la următoarea permutare p' înseamnă o parcurgere a frunzelor acestui arbore.

Fie $p = (p(1), \ldots, p(n))$; atunci p' se construieşte astfel:

Dacă n apare pe o poziţie $m > 1$, atunci se face interschimbarea $p(m) \leftrightarrow p(m-1)$ şi s-a obţinut noua permutare. Dacă $p(1) = n$, atunci se permută circular spre stânga cele n elemente, obţinându–se tatăl permutării p. În continuare, dacă $n-1$ apare pe o poziţie $m \neq 1$, atunci se efectuează interschimbarea $p(m) \leftrightarrow p(m-1)$ şi s–a obţinut noua permutare; în caz contrar, se permută circular ultimele $n-1$ poziţii şi se încearcă deplasarea spre stânga a lui $n-2$, etc. Algoritmul este:

Perm2(p, n, ig)
dacă $ig = 0$ *atunci*
  pentru $i = 1, n$
    $p(i) = i$
  sfârşit pentru

$ig = 1$
Return
sfârşit dacă
pentru $i = n, 2, -1$
$m = 1$
cât timp $p(m) \neq i$
$m = m + 1$
sfârşit cât timp
dacă $m \neq 1$ *atunci*
$p(m-1) \leftrightarrow p(m)$
Return
sfârşit dacă
cheamă PermCircStanga(p, n, i)\{ se permută circular primele i componente ale lui P\}
Return
sfârşit pentru
$ig = 0$
Return

unde PermCircStanga(p, n, i) este următorul algoritm:

PermCirc(p, n, i)
$x = p(1)$
pentru $j = 1, i - 1$
$p(j) = p(j + 1)$
sfârşit pentru
$p(i) = x$
Return

7.7 Generarea mulţimii aranjamentelor

O primă metodă constă în generarea tuturor combinărilor de n luate câte m şi pentru fiecare din acestea realizarea tuturor permutărilor. Această idee duce la următorul algoritm:

Aranj1$(x, n, m, ig, c, p, ig1)$
dacă $ig = 0$ *atunci*
pentru $i = 1, m$
$x(i) = i$
$c(i) = i$
$p(i) = i$
sfârşit pentru
$ig = 1$

$ig1 = 1$
Return
sfârşit dacă
cheamă Perm$(p, m, ig1)$
dacă $ig1 \neq 0$ *atunci*
pentru $i = 1, m$
$x(i) = c(p(i))$
sfârşit pentru
altfel
cheamă Combin(c, m, ig)
dacă $ig = 1$ *atunci*
pentru $i = 1, m$
$p(i) = i$
$x(i) = c(i)$
sfârşit pentru
$ig1 = 1$
Return
sfârşit dacă
sfârşit dacă
Return

Să observăm că metoda nu generează aranjamentele în ordine lexicografică crescătoare. Următoarea metodă urmăreşte să păstreze această ordine, conform strategiei generale Backtracking.

Fie $x = (x(1), \ldots, x(m))$ un aranjament oarecare. Pentru determinarea lui x', aranjamentul succesor, în ordine lexicografică, vom determina mai întâi indicele i, cel mai mare indice care poate fi mărit. Un $x(i)$ nu poate fi mărit dacă valorile $x(i)+1$, $x(i)+2$, $\ldots$, n nu sunt disponibile. Vom folosi un vector $disp$ cu n componente care sunt 0 sau 1. $disp(k) = 0$ arată că valoarea k este disponibilă iar $disp(k) = 1$ arată că valoarea $x(k)$ nu este disponibilă (este deja folosită în x). Când se găseşte valoarea x cu proprietatea menţionată, $x(i)$, $x(i+1)$, $\ldots$, $x(m)$ vor primi cele mai mici valori disponibile, în ordine crescătoare. Rezultă de aici că până la găsirea lui i toate valorile cercetate regresiv vor fi făcute disponibile.

Obţinem în acest fel următorul algoritm:

Aranj2$(x, n, m, ig, disp)$
dacă $ig = 0$ *atunci*
pentru $i = 1, m$
$x(i) = i$
$disp(i) = 1$
sfârşit pentru

```
    pentru i = m + 1, n
      disp(i) = 0
    sfârşit pentru
    ig = 1
    Return
  sfârşit dacă
  pentru i = m, 1, −1
    disp(x(i)) = 0
    pentru j = x(i) + 1, n
      dacă disp(j) = 0 atunci
        x(i) = j
        disp(j) = 1
        k = 0
        pentru l = i + 1, m
          repetă
            k = k + 1
          până când disp(k) = 0
          x(l) = k;
          disp(k) = 1
        sfârşit pentru
        Return
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
  ig = 0
  Return
```

7.8 Generarea submulţimilor de sumă dată

Fie $A = (a(1), \ldots, a(n))$ un vector cu componente strict pozitive şi $M > 0$ un număr pozitiv dat. Se cer submulţimile $B = \{i_1, i_2, \ldots, i_l\} \subseteq \{1, 2, \ldots, n\}$ astfel încât

$$\sum_{i \in B} a(i) = M \tag{7.2}$$

Vom folosi pentru reprezentarea mulţimilor B metoda 3, deci dacă x este vectorul caracteristic al lui B atunci (7.2) se rescrie ca:

$$\sum_{i=1}^{n} x(i) \cdot a(i) = M \tag{7.3}$$

Componentele lui x vor fi generate succesiv după metoda generală backtracking: dacă am stabilit deja în x componentele $x(1), x(2), \ldots, x(k-1)$, $\sum_{i=1}^{k-1} x(i) \cdot a(i) \leq M$, atunci componenta $x(k)$ trebuie să satisfacă condiţiile:

$$\sum_{i=1}^{k} x(i) \cdot a(i) \leq M \tag{7.4}$$

şi

$$\sum_{i=1}^{k} x(i) \cdot a(i) + \sum_{i=k+1}^{n} a(i) \geq M \tag{7.5}$$

Vom considera

$$S = \sum_{i=1}^{k} x(i) \cdot a(i) \tag{7.6}$$

şi

$$R = \sum_{i=k+1}^{n} a(i) \tag{7.7}$$

Vom presupune că elementele mulţimii A sunt sortate crescător (se face o sortare în prealabil). Algoritmul va fi:

SumbultimiSuma$(a, x, n, M, k, S, R, ig)$
dacă $ig = 0$ *atunci*
  pentru $i = 0, n$
    $x(i) = -1$
    $k = 1$
    $S = 0$
    $R = \sum_{i=1}^{n} a(i)$
  sfârşit pentru
altfel
  $S = S - x(k) \cdot a(k)$
  $R = R + a(k)$
  $x(k) = -1$
  $k = k - 1$
sfârşit dacă
cât timp $k > 0$
  dacă $x(k) < 1$ *atunci*
    $x(k) = x(k) + 1$
    $S = S + x(k) \cdot a(k)$
    dacă $S = M$ *atunci*
      $ig = 1$

```
        Return
      sfârşit dacă
      dacă S < M şi S + R ≥ M atunci
        R = R − x(k)
        k = k + 1
      sfârşit dacă
      dacă S + R < M atunci
        S = S + x(k) · a(k)
        R = R + a(k)
        x(k) = −1
        k = k − 1
      sfârşit dacă
    altfel
      S = S − a(k)
      R = R + a(k)
      x(k) = −1
      k = k − 1
    sfârşit dacă
  sfârşit cât timp
  ig = 0
```

Varianta recursivă, mult mai uşor de urmărit este dată mai jos. Variabila $ramas$ este egala cu $M - S$ din algoritmul precedent.

```
SubmultimiRecursiv(a, x, n, k, ramas)
dacă ramas = 0 atunci
  cheamă Prelucrare(a, x, k − 1)
  Return
sfârşit dacă
dacă k > n sau ramas < 0 atunci
  Return
sfârşit dacă
pentru x[k] = 0, 1
  cheamă SubmultimiRecursiv(a, x, n, k + 1, ramas − x[k] · a[k])
sfârşit pentru
Return
```

7.9 Rezumat

Pentru determinarea unor submulţimi (de o natură particulară) a unei mulţimi există algoritmi specializaţi. Se pot da, evident, mai mulţi algoritmi pentru generarea unei soluţii. Aceste tipuri de probleme se întâlnesc adeseori

sub o formă mascată în enunţul altor probleme, sau ca etapă intermediară într–un caz mai mare.

7.10 Exerciţii (rezolvările la pagina 160)

1. Să se scrie algoritmi recursivi pentru generarea tipurilor de submulţimi din acest capitol.

Capitolul 8

Metoda *Divide et Impera*

8.1 Obiective

Capitolul prezintă metoda *Divide et Impera*, ce se poate utiliza atunci când rezolvarea unei probleme se face prin "spargerea" ei în cazuri de dimensiune mai mică (dar de aceeaşi natură cu problema iniţială), iar apoi prin recombinarea soluţiilor obţinute. Determinarea complexităţii se reduce la rezolvarea unei recurenţe, pentru care o unealtă utilă este Teorema centrală. Este dat cel mai eficient algoritm de sortare (Quicksort) şi se demonstrează o limită inferioară pentru algoritmiide sortate bazaţi pe comparaţii.

8.2 Prezentare generală

Această metodă nu este ceva nou pentru cursul nostru. Am rezolvat cu ea problema sortării prin interclasare în capitolul 3, pagina 21. Metoda constă în următoarele etape:

1. Partiţionarea problemei iniţiale în mai multe probleme mai mici;
2. Rezolvarea (de obicei recursivă) a fiecărei subprobleme; dacă o subproblemă este de dimensiune suficient de mică, atunci ea se poate rezolva printr–o metodă "ad-hoc", care poate fi mai puţin performantă pentru cazuri de dimensiune mare;
3. Combinarea rezultatelor parţiale pentru a obţine rezultatul problemei iniţiale.

Dacă o problemă a fost divizată în a subprobleme care au dimensiunea $\frac{n}{b}$ şi sunt rezolvate recursiv, atunci complexitatea este dată de formula:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{8.1}$$

unde $f(n)$ este costul combinării soluţiilor subproblemelor.

Aflarea formei lui $T(n)$ este o problemă de algebră care se poate aborda prin inducţie sau folosind rezultatele teoremei principale din [2]:

Teorema 1. *(Teorema centrală.) Fie $a \geq 1$ şi $b > 1$ constante, fie $f(n)$ o funcţie şi $T(n)$ definită pe întregii nenegativi prin recurenţa:*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{8.2}$$

unde interpretăm pe $\frac{n}{b}$ fie ca pe $\lceil \frac{n}{b} \rceil$, fie ca pe $\lfloor \frac{n}{b} \rfloor$. Atunci $T(n)$ poate fi delimitată asimptotic după cum urmează:

1. *Dacă $f(n) = O(n^{\log_b a - \epsilon})$ pentru o anumită constantă $\epsilon > 0$, atunci $T(n) = \Theta(n^{\log_b a})$*
2. *Dacă $f(n) = \Theta(n^{\log_b a})$ atunci $T(n) = \Theta(n^{\log_b a} \log n)$*
3. *Dacă $f(n) = \Omega(n^{\log_b a + \epsilon})$ pentru o anumită constantă $\epsilon > 0$ şi dacă $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru o anumită constantă $c < 1$ şi toţi n suficienţi de mari, atunci $T(n) = \Theta(f(n))$.*

Demonstraţia se poate găsi în [2].

Am folosit *divide et impera* la sortarea unui şir (sortarea prin interclasare, algoritmul *Mergesort*); se vede că se ajunge la complexitatea:

$$T(n) = \Theta(n \log_2 n)$$

(care se deduce din cazul 2 al Teoremei centrale), complexitate mai bună decât cea de la sortarea prin inserţie, care era

$$T(n) = \Theta(n^2)$$

8.3 Problema turnurilor din Hanoi

Prima problemă pe care o prezentăm este "Problema turnurilor din Hanoi", problemă care nu are nici o legătură cu Hanoi-ul, ci este un joc logic inventat în secolul trecut de matematicianul francez Eduard Lucas. El a fost cel care a vândut pentru prima oară această problemă drept joc, cu opt discuri şi trei baghete.

După o străveche legendă hindusă, zeul Brahma ar fi fixat pe masa templului său din Benares trei bastonaşe verticale de diamant şi ar fi înşirat pe acesta 64 de discuri de mărimi descrescătoare (de la bază la vârf), formând

astfel un trunchi de con. A dispus apoi ca discurile să fie mutate pe un al doilea, folosind şi al treilea bastonaş; zi şi noapte trebuia mutat câte unul din discurile aflate în vârful unuia din bastonaşe în vârful altui bastonaş, respectând în fiecare moment ordinea descrescătoare a mărimii discurilor aflate pe fiecare baştonaş. Legenda spune că, dând această poruncă, Brahma a susţinut că atunci când se va termina aşezarea celor 64 de dicuri pe cel de-al doilea baştonaş, atunci se va sfârşi şi viaţa pe Pământ. O versiune destul de optimistă deoarece presupunând că mutarea unui disc ar dura o secundă, cele 64 de discuri vor ajunge pe al doilea bastonaş în circa 584.942.416.729 de ani.

Enunţul formalizat al problemei este: fie trei tije şi n discuri perforate de diametre diferite. Discurile sunt aşezate iniţial pe tija 1 în ordinea descrescătoare a diametrelor acestora considerând sensul de la bază la vârf. Problema constă în a muta "turnul" de n discuri de pe tija 1 pe tija 2 într-un număr minim de mutări, ţinând cont de următoarele reguli:

1. La fiecare mutare se mută un singur disc
2. În permaneţă, pe fiecare tijă, deasupra unui disc pot fi mutate numai discuri de diametre mai mici.

Calcularea numărului minim de paşi: observăm că pentru 2, 3, 4 şi 5 discuri, problema se rezolvă efectuând un număr de mutări egal cu 3, 7, 15 şi respectiv 31, de unde putem formula ipoteza că pentru problema cu n discuri vom efectua $2^n - 1$ mutări.

Să demonstrăm valabilitatea acestui rezultat prin inducţie matematică completă după n. Am arătat că afirmaţia este adevărată pentru $n = 2, 3, 4, 5$ discuri. Presupunem că ea este adevărată pentru n discuri, deci sunt necesare $2^n - 1$ mutări. Vom demonstra că pentru $n+1$ discuri sunt necesare $2^{n+1} - 1$ mutări.

Putem rezolva problema mutării celor $n + 1$ discuri în trei etape:

1. Mutăm primele n discuri pe tija a treia
2. Mutăm al $n + 1$-lea disc pe a doua tijă
3. Mutăm cele n discuri de pe a treia tijă pe cea de-a două tijă.

Folosind acest procedeu, sunt necesare:

1. $2^n - 1$ mutări pentru realizarea primei etape
2. o mutare pentru realizarea celei de a doua etape

3. $2^n - 1$ mutări pentru realizarea celei de a treia etape

În total sunt suficiente $2^{n+1} - 1$ mutări În continuare vom prezenta mai mulţi algoritmi, fiecare bazându–se pe câte o observaţie importantă.

8.3.1 Metoda I

O mutare poate fi scrisă ca o pereche (i, j) cu $i \in \{1, 2, 3\}$ şi $i \neq j$, semnificând că se mută discul cel mai mic de pe tija i pe tija j. Mai notăm cu $H(m, i, j)$ şirul mutărilor necesare pentru a muta primele m discuri (cele mai de sus) de pe tija i pe tija j. În aceste ipoteze, problema se reduce la a determina şirul de mutări $H(m, i, j)$. Observăm că putem scrie:

$$H(m,i,j) = \begin{cases} \text{(i,j)}, & \text{pentru } m = 1 \\ \text{H(m-1, i, k),(i,j),H(m-1, k, j)} & \text{pentru } m \neq 1 \end{cases} \tag{8.3}$$

unde $k = 6 - i - j$ este tija diferită de tijele i şi j.

Cu aceasta reducem rezolvarea problemei cu n discuri la rezolvarea a două probleme cu $n - 1$ discuri. Algoritmul se bazează pe ideea că în loc de $H(m, i, j)$ putem scrie $H(i, j)$, memorând valoarea lui m separat. Se mai poate observa că $H(i, j)$ se transformă în $H(i, k)$, (i, j), $H(k, j)$ cu $k = 6 - i - j$. Având în mijloc tot perechea (i, j), rezultă că imediat ce o pereche este generată, ea poate fi înscrisă direct pe locul ei. Algoritmul obţine cele $2^n - 1$ mutări folosind în acest scop o matrice M cu 2 linii şi $2^n - 1$ coloane, fiecare coloană memorând o mutare. Începem prin a înscrie perechea (1,2) pe coloana 2^{n-1}. Perechile ce vor lua naştere din ea vor fi înscrise pe coloanele 2^{n-2} şi $2^{n-1} + 2^{n-2}$. Se observă că acest algoritm face parte din clasa algoritmilor "Divide et Impera".

Mai general, pentru fiecare valoare a lui $m \in \{n, n-1, \ldots, 2\}$ se expandează perechile din coloanele c de forma $c = \mathcal{M}2^m + 2^{m-1}$[1], perechile care rezultă din cea din coloana c fiind înscrise pe coloanele $c - 2^{m-2}$, $c + 2^{m-2}$.

Hanoi1$(M, n, N)\{N = 2^n - 1\}$
$k1 = N + 1$
$k2 = k1/2$
$k3 = k2/2$
$M(1, k2) = 1$
$M(2, k2) = 2$
pentru $m = n, 2, -1$
  pentru $l = k2, N, k1$
    $i = M(1, l)$

[1] $\mathcal{M}p$ este notaţie pentru "multiplu de p".

$j = M(2, l)$
$k = 6 - i - j$
$M(1, l - k3) = i$
$M(2, l - k3) = k$
$M(1, l + k3) = k$
$M(2, l + k3) = j$
sfârşit pentru
$k1 = k2$
$k2 = k3$
$k3 = k3/2$
sfârşit pentru

8.3.2 Metoda a II–a

Varianta recursivă este bazată tot pe metoda "Divide et Impera". Această metodă este elegantă din punct de vedere al redactării, dar ineficientă din cauza spaţiului ocupat în stivă şi a întârzierilor datorate apelurilor în cascadă a procedurii.

Hanoi2(n, i, j)
dacă $n = 1$ *atunci*
  mută discul superior de pe tija i pe tija j
altfel
  cheamă Hanoi2$(n - 1, i, 6 - i - j)$
  mută discul superior de pe tija i pe tija j
  cheamă Hanoi2$(n - 1, 6 - i - j, j)$
sfârşit dacă
Return

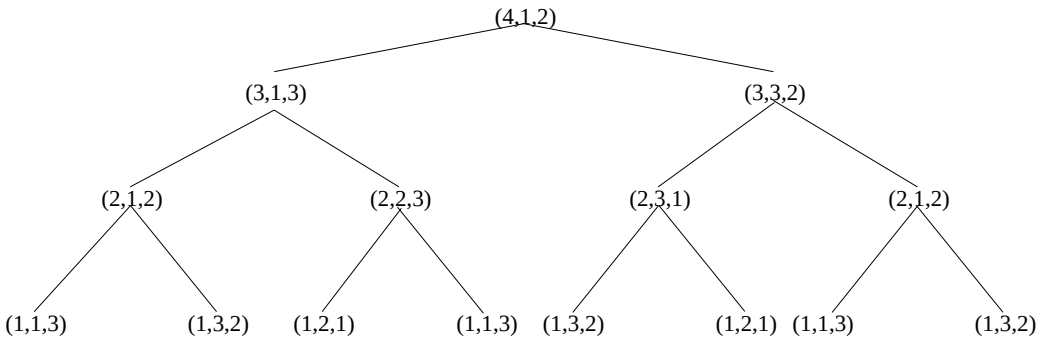


Figura 8.1: Arborele apelurilor recursive pentru Hanoi2(4, 1, 2)

8.3.3 Metoda a III–a

Vom porni de la algoritmul precedent, pe care îl vom transforma într–unul iterativ. Apelul Hanoi2(4,1,2) va duce la arborele de apeluri recursive din figura 8.1, unde în fiecare nod este memorat un triplet de forma (n, i, j).

Prin parcurgerea în inordine a acestui arbore binar şi afişarea pentru fiecare nod a ultimelor două numere memorate în el, se ajunge la determinarea soluţiei cerute. Pentru parcurgere se va folosi o stivă S în care se vor memora triplete de forma (a, b, c). Arborele nu este construit în mod explicit.

Hanoi3(n)
$i = 1$
$j = 2$
$gata = fals$
$niv = 1$\{pornim de la rădăcină, care se află pe nivelul 1\}
$S = \emptyset$\{S este stivă\}
$con = 0$
repetă
  cât timp $niv < n - 1$
    $niv = niv + 1$
    $(i, j, niv) \Rightarrow S$
    $j = 6 - i - j$
  sfârşit cât timp
  scrie($i, 6 - i - j$)
  scrie(i, j)
  scrie($6 - i - j, j$)
  dacă $S = \emptyset$ *atunci*
    $gata = adevarat$
  altfel
    $(i, j, niv) \Leftarrow S$
    scrie(i, j)
    $i = 6 - i - j$
  sfârşit dacă
până când $gata = adevarat$
Return

Observaţie: Pentru implementarea stivei se poate folosi o matrice cu $n - 2$ linii şi 3 coloane.

8.4 Algoritmul de sortare QuickSort

Fie şirul $x = (x(1), x(2), \ldots, x(n))$. Dorim să sortăm acest şir, adică $\forall i \in \{1, \ldots, n-1\}$, $x(i) \leq x(i+1)$. După cum ştim, algoritmii sunt caracterizaţi de timpul de execuţie şi spaţiul de memorie ocupat.

Sortarea unui şir cu metoda bulelor sau cea a inserţiei s-a făcut cu algoritmi "in place" (în afara şirului s–au menţinut cel mult un număr constant de elemente din şir), iar prin metoda interclasării se ocupă un alt şir (nu este un algoritm "in place") şi în plus timpul de execuţie ascunde o constantă multiplicativă mare.

Ne dorim să realizăm un algoritm de sortare prin metoda "Divide et Impera", fără a folosi un şir intermediar, deci "in place". Profesorul C.A.R. Hoare de la Universitatea Oxford a propus împăţirea şirului în părţi nu neapărat egale având proprietatea că toate elemetele primei părţi să fie mai mici sau egale decât un $x(k)$ fixat (determinat de algoritmul de partitionare care va fi prezentat mai jos), iar componentele celei de a doua părţi să fie mai mari decât $x(k)$. Faza de combinare lipseşte (este o metodă "Divide et Impera" şchioapă).

Algoritmul QuickSort (sortare rapidă) este:

QuickSort(x, p, r)
dacă $p < r$ *atunci*
  cheamă Part(x, p, r, k)
  cheamă QuickSort(x, p, k)
  cheamă QuickSort$(x, k+1, r)$
sfârşit dacă
Return

Algoritmul care realizează partiţionarea este:

Part(x, p, r, k)
$pivot = x(p)$
$i = p - 1$
$j = r + 1$
$terminat = fals$
repetă
  repetă
    $j = j - 1$
  până când $x(j) \leq pivot$
  repetă
    $i = i + 1$
  până când $x(i) \geq pivot$
  dacă $i < j$ *atunci*

$x(i) \leftrightarrow x(j)$
altfel
$k = j$
$terminat = adevarat$
sfârşit dacă
până când $terminat = adevarat$
Return

Exemplu: fie şirul:

	5	3	2	6	4	1	3	7	
i	$\rightarrow$							$\leftarrow$	j

Avem: $pivot = 5$, $p = 1$, $r = 8$. Primul element al şirului mai mic sau egal decât pivotul 5 este 3 (în căutarea de la dreapta la stânga) şi primul element al şirului mai mare sau egal decât pivorul 5 este 5 (căutare de la stânga spre dreapta). După găsirea lor le interschimbăm, ajungând la configuraţia:

3	3	2	6	4	1	5	7
i	$\rightarrow$				$\leftarrow$	j	

Apoi, succesiv:

3	3	2	1	4	6	5	7
			i	$\rightarrow\leftarrow$	j		

3	3	2	1	4	6	5	7
			$\leftarrow$	j	i	$\rightarrow$	

Deoarece j a devenit mai mic decât i, se iese din procedura de partiţionare cu valoarea lui $k = 5$. În acest moment s–a terminat partiţionarea după $pivot = 5$. În continuare se face partiţionarea şirului din partea stângă pentru $pivot = 3$:

	3	3	2	1	4	
i	$\rightarrow$				$\leftarrow$	j

1	3	2	3	4
i	$\rightarrow$	$\leftarrow$	j	

1	2	3	3	4
	$i \rightarrow$	$\leftarrow j$		

1	2	3	3	4
	$\leftarrow j$	$i \rightarrow$		

Deoarece avem $j \geq i$, procedura se termină valoarea lui $k = 2$. Analog se procedează pentru fiecare subcaz, până când se ajunge la vectori de lungime 1, care sunt (evident) sortaţi.

Observaţie: Când şirul este deja ordonat crescător, se face partiţionare după fiecare element.

Complexitatea partiţionării este dată de $T_P(n) = c \cdot n$, c constantă. Complexitatea QuickSort în cazul şirului iniţial sortat este:

$$\begin{aligned}
T_Q(n) &= T_Q(1) + T_Q(n-1) + T_P(n) = \\
&= T_Q(n-1) + T_P(n) + c = \\
&= T_Q(1) + T_Q(n-2) + T_P(n-1) + T_P(n) + c = \\
&= T_Q(n-2) + T_P(n-1) + T_P(n) + 2c = \\
&= \cdots = \\
&= T_Q(2) + \sum_{k=3}^{n} T_P(k) + (n-2)c = \\
&= T_Q(1) + T_Q(1) + \sum_{k=2}^{n} T_P(k) + (n-2)c = \\
&= \sum_{k=2}^{n} T_P(k) + cn = \sum_{k=2}^{n} c \cdot k + c \cdot n = \Theta(n^2)
\end{aligned}$$

unde am considerat în mod convenabil că $T_Q(1)$ este mărginit de aceeaşi constantă c care apare în expresia lui $T_P(n)$.

În cel mai bun caz, când şirul se împarte în jumătate, $T_Q(n) = 2T_Q(n/2) + \Theta(n) = \Theta(n \log n)$ (conform teoremei centrale). Se poate demonstra că timpul mediu al QuickSort–ului este $\Theta(n \log n)$. În practică se dovedeşte că QuickSort este *cel mai rapid algoritm de sortare!!*

Ne punem întrebarea dacă există algoritmi de sortare cu complexitatea mai mică decât $n \log n$? Se poate demonstra că dacă algoritmul se bazează doar pe comparaţii între elemente, nu se poate coborî sub bariera de $\Theta(n \log n)$. Pentru a demonstra, alegem a_1, a_2, a_3 elemente pe care le comparăm (Figura 8.2). Se obţine un arbore binar care are drept frunze permutările indicilor elementelor iniţiale, adică 3! frunze.

În cazul a n valori, obţinem un arbore binar cu $n!$ frunze. Prin înălţimea unui arbore înţelegem lungimea drumului maxim de la rădăcină la orice frunză şi se notează cu h. h reprezintă numărul de comparaţii în cazul cel mai defavorabil. Este evident că numărul maxim de frunze pentru un arbore binar de adâncime h este 2^h. Deci în cazul nostru pentru compararea a n elemente vom avea un arbore cu o adâncime h astfel încât $n! \leq 2^h$. Rezultă

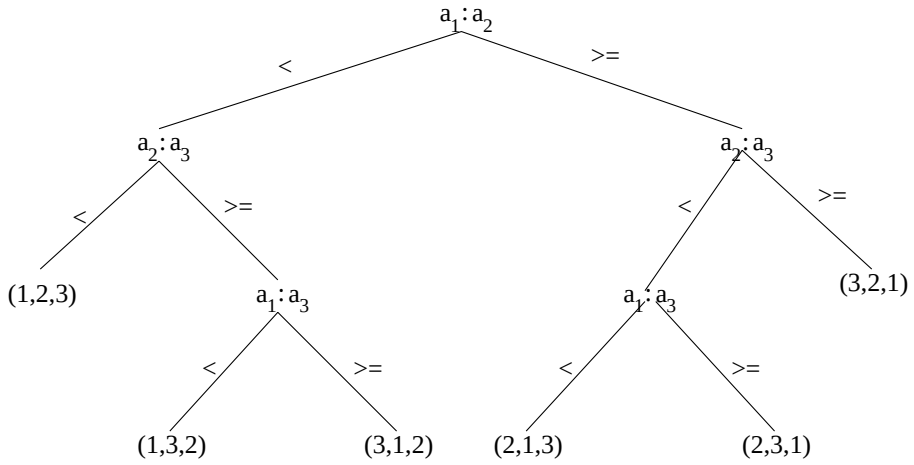


Figura 8.2: Arborele de decizie pentru 3 elemente

că $h \geq \log_2(n!)$ şi folosind aproximaţia lui Stirling:

$$n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n {}^2$$

obţinem că:

$$h \geq \log_2 n! > n\log_2 n - n\log_2 e = \Omega(n\log_2 n)$$

Se obţine că $T(n) = \Omega(n\log_2 n)$. Am demonstrat deci că pentru algoritmii care se bazează doar pe comparaţii complexitatea nu poate să scadă sub $n\log_2 n$. Pentru cazuri speciale ale şirului ce trebuie sorart, a se vedea capitolul 9.

8.5 Rezumat

Metoda Divide et Impera se foloseşte atunci când rezolvarea unei probleme se poate face prin descompunerea ei în cazuri de dimensiune mai mică, rezolvarea acestora dupa acelaşi principiu, apoi combinarea rezultatelor lor. Deseori algoritmii de acest tip se scriu ca recursiv (datorită naturii lor), iar determinarea complexităţii se face prin rezolvarea unei ecuaţii recursive, a cărei soluţii se poate da de multe ori pe baza Teoremei centrale.

Un exemplu important de algoritm de tip divide et Impera este Quicksort. Problemele de la sfârşitul capitolului dau şi alte exemple ale utilităţii metodei.

[2]Se poate arăta că: $\sqrt{2\pi}n^{n+\frac{1}{2}}e^{-n}\cdot e^{(12n+1)^{-1}} < n! < \sqrt{2\pi}n^{n+\frac{1}{2}}e^{-n}\cdot e^{(12n)^{-1}}$

8.6 Exerciţii (rezolvările la pagina 162)

1. Bazându–vă pe observaţia că mutarea din mijloc se ştie, scrieţi un algoritm iterativ pentru problema turnurilor din Hanoi, punând mereu mijloacele intervalelor obţinute prin înjumătăţire într-un şir cu numărul cunoscut de mutări.

2. Având la intrare un şir ordonat crescător, să se scrie un algoritm de tip "Divide et Impera" pentru găsirea poziţiei unui element de valoare dată x, dacă el există. Se va folosi metoda înjumătăţirii intervalului dat de numărul de elemente al şirului. Care este complexitatea căutării?

3. Rezolvaţi aceeaşi problemă prin căutare secvenţială (testarea pe rând a elementelor). Care este în acest caz complexitatea?

4. Demonstraţi că timpul de execuţie al algoritmului QuickSort, în cazul unui vector cu toate elementele egale, este $\Theta(n \log n)$.

5. Complexitatea algoritmului QuickSort este mare din cauză că în cazuri defavorabile, partiţionarea se face cu un singur element într-una din partiţii. Rescrieţi algoritmul de partiţionare, luând ca element de separare nu primul element, ci valoarea din mijloc între primul element, ultimul şi cel aflat pe poziţia din mijloc.

Capitolul 9

Sortare şi statistici de ordine

9.1 Obiective

În acest capitol ne propunem să rezolvăm următoarele probleme, relative la un şir cu n elemente:

1. să se găsească a i-a componentă în ordine statistică a şirului
2. să se găsească mediana (componenta din mijloc) unui şir
3. să se găsească cele mai mici două elemente dintr–un şir
4. determinarea simultană a minimului şi maximului

Observaţie: Vom considera că şirul este cu elemente distincte în rezolvarea tuturor problemelor enunţate, urmând apoi să extindem rezultatele obţinute şi pentru situaţia în care elementele se repetă.

Vor fi consideraţi mai mulţi algoritmi care rezolvă această problemă. Capitolul conţine şi prezentări ale altor algoritmi de sortare (heapsort, cu care ocazie se introduce şi structura de date numită heap, sortări în timp liniar).

9.2 Exemplificări, generalităţi

Exemple: Fie $x = (5, 4, 2, 8, 9, 1, 7)$. Ce se înţelege prin a i-a componentă în ordine statistică? Avem $n = 7$ şi în mod evident $1 \leq i \leq 7$. Dacă îl consider pe $i = 3$, vom înţelege prin a i-a componentă în ordine statistică a şirului x elementul al treilea din şirul x sortat în ordine crescătoare.

$$x = (1, 2, 4, 5, 7, 8, 9), \quad x(3) = 4$$

Mediana depinde de paritatea numărului de elemente din şirul dat. Dacă n este impar atunci mediana este unică şi se găseşte în şirul sortat pe poziţia $i = (n+1)/2$. Altfel avem de–a face cu două mediane, una dintre ele pe poziţia $i = n/2$, cealaltă pe poziţia $i = n/2 + 1$. În general, pentru un şir de lungime n vom considera mediană elementul de pe poziţia $\left[\frac{n+1}{2}\right]$ din şirul sortat. În cazul nostru mediana este 5.

Ce reprezintă minimul şi maximul din perspectiva statisticilor de ordine? Minimul este prima componentă în ordine statistică a unui şir. Maximul este a n-a componentă în ordine statistică.

9.3 Determinarea simultană a minimului şi a maximului

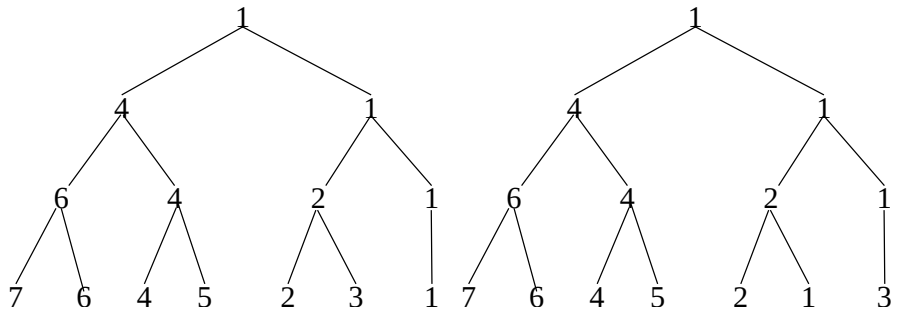
MinMax(x, n)
dacă $n \mod 2 = 0$ *atunci*
  dacă $x(1) < x(2)$ *atunci*
    $min = x(1)$
    $max = x(2)$
  altfel
    $min = x(2)$
    $max = x(1)$
  sfârşit dacă
  $k = 3$
altfel
  $min = x(1)$
  $max = x(1)$
  k=2
sfârşit dacă
pentru $i = k, n-1, 2$
  dacă $x(i) < x(i+1)$ *atunci*
    $mic = x(i)$
    $mare = x(i+1)$
  altfel
    $mic = x(i+1)$
    $mare = x(i)$
  sfârşit dacă
  dacă $mic < min$ *atunci*
    $min = mic$
  sfârşit dacă

dacă $mare > max$ *atunci*
$max = mare$
sfârşit dacă
sfârşit pentru
scrie("minimul este ", min)
scrie("maximul este ", max)
Return

Numărul de comparaţii între elemente necesare în determinarea independentă a min şi max este $2(n-1)$ (prin două parcurgeri ale şirului), în schimb pentru determinarea simultană a min şi max avem nevoie de $3\lceil n/2 \rceil - 2$ comparaţii (demonstraţi acest lucru). Sugerăm cititorului să execute algoritmul pentru intrarea $x = (1, 2, -3, 4, 5, 6, -85, 2)$.

9.4 Găsirea celor mai mici două elemente dintr–un şir

Ideea algoritmului prezentat mai jos este următoarea: se compară elementele 1 cu 2, 3 cu 4, etc; pentru fiecare pereche comparată se determină cel mai mic. Aceste elemente "câştigătoare" se vor compara la rândul lor pe perechi adiacente, ca mai sus. Când rămâne un singur element, acesta este minimul şirului. Ataşăm acestor operaţii un arbore binar (posibil nestrict) echilibrat, construit în felul următor: există n noduri terminale (frunze), fiecare conţinând drept informaţie câte un element din şir; un nod neterminal are drept valoare minimul din cei doi fii (figura 9.4).



(a) Arborele de comparaţii pentru vectorul $(7, 6, 4, 5, 2, 3, 1)$

(b) Arborele de comparaţii pentru vectorul $(7, 6, 4, 5, 2, 1, 3)$

Figura 9.1: Exemple de arbori construiţi conform strategiei de mai sus.

Evident că pentru fiecare nod neterminal, informaţia sa coincide cu a unui descendent al său. În cadrul acestui algoritm vom folosi termenul "învins" în exprimarea "x este învins de y" dacă $x > y$ (deoarece se caută minimul). Pentru determinarea celui de al doilea minim trebuie să determinăm cel mai mic element din vectorul iniţial "învins" de minim: în exemplele date este 2, "învins" de 1. Pentru aceasta vom considera informaţia din cei doi descendenţi ai rădăcinii, în ambele exemple date fiind 4 şi 1. 4 este cel mai mic element din jumătatea stângă a şirului (este descendent stâng al rădăcinii) şi va trebui să căutăm în jumătatea dreaptă a şirului cel mai mic număr, învins de rădăcină; acest lucru se face (în exemplele date) coborând în subarborele drept spre frunze. Se compară toate elementele pe care informaţia din rădăcină le–a depăşit cu minimul din jumătatea stângă (în cazul nostru 4); coborârea se termină atunci când fie se ajunge la frunze (cazul 9.1(b)), fie este evident că nu mai are sens coborârea (cazul 9.1(a)).

În algoritmul de mai jos nu se va construi explicit arborele. Se vor folosi nişte cozi pentru menţinerea pentru fiecare element din şir a elementelor "învinse" de acesta.

MinMin($x, n, min1, min2$)
pentru $i = 1, n$
  $C(i) = \emptyset$ {$C(i)$ este coadă}
sfârşit pentru
cât timp $n > 1$
  $pozitie = 1$ {$pozitie$ este indicele din vector unde se vor pune elementele "învingătoare"}
  pentru $i = 1, n-1, 2$
    dacă $a(i) < a(i+1)$ *atunci*
      $imin = i$
      $imax = i + 1$
    altfel
      $imin = i + 1$
      $imax = i$
    sfârşit dacă
    $a(pozitie) = a(imin)$
    $C(pozitie) = C(imin) \cup \{a(imax)\}$
    $pozitie = pozitie + 1$
    dacă $n \ mod \ 2 \neq 0$ *atunci*
      {a rămas un element care nu are cu cine se compara}
      $a(pozitie) = a(n)$
      $C(pozitie) = C(n)$
      $pozitie = pozitie + 1$

sfârşit dacă
$n = pozitie - 1$
sfârşit pentru
sfârşit cât timp
$min1 = a(1)$
$min2 \Leftarrow C(1)$
cât timp $C(1) \neq \emptyset$
$y \Leftarrow C(1)$
dacă $y < min2$ *atunci*
$min2 = y$
sfârşit dacă
sfârşit cât timp
Return

Prin $C(i) = C(j)$ înţelegem trecerea elementelor din coada $C(j)$ în coada $C(i)$ (dacă cozile sunt implementate folosind liste înlănţuite, atunci această operaţie se poate face în timp $\Theta(1)$), iar prin $C(i) = C(j) \cup \{x\}$ se înţelege $x \Rightarrow C(j)$ urmat de $C(i) = C(j)$.

Numărul de comparaţii se determină după cum urmează: $n - 1$ pentru determinarea minimului (terminarea ciclului *cât timp*), după care numărul de comparaţii pentru determinarea minimului din coada $C(1)$. Cum în această coadă există cel mult $\lceil \log_2 n \rceil$ elemente (înălţimea arborelui de comparaţii - demonstraţia o lăsăm cititorului), avem că se execută cel mult $\lceil \log_2 n \rceil - 1$ comparaţii pentru determinarea celui de al doilea minim. Deci în total avem $n + \lceil \log_2 n \rceil - 2$ comparaţii în cazul cel mai defavorabil.

9.5 Problema selecţiei

9.5.1 Selecţie în timp mediu liniar

În general determinarea celei de a i–a componente este o problemă având ca date de intrare un şir a cu n elemente distincte şi un număr i cu proprietatea $1 \leq i \leq n$, iar ca dată de ieşire un element rez al şirului a care are proprietatea că există exact $i - 1$ elemente ale şirului mai mici decât rez.

Sel(A, p, r, i, rez)
dacă $p = r$ *atunci*
$rez = a(p)$
Return
sfârşit dacă
Part(A, p, r, k)\{Part este algoritmul de partiţionare din QuickSort\}
dacă $i \leq k$ *atunci*

Sel(a, p, k, i, rez)
altfel
Sel$(a, k+1, r, i-k, rez)$
sfârşit dacă
Return

Algoritmul Sel aplicat unui şir deja sortat are (cel mai rău caz) complexitatea $O(n^2)$. Se poate arăta (vezi lucrările [3], [2]) că pentru cazul mediu complexitatea este de $O(n)$.

9.5.2 Selecţia în timp liniar în cazul cel mai defavorabil

Ideea algoritmului următor este de a obţine o parţitionare mai bună decât îversiunea precedentă a algoritmului. Vom folosi algoritmul Part modificat astfel încât să preia ca parametru de intrare elementul în jurul căruia este efectuată partajarea.

Algoritmul Selectie modificat determină al i–lea cel mai mic element al unui tablou de intrare de n elemente, efectuând următorii paşi (oprirea algoritmului se face atunci când subşirul pe care se face căutarea are un element, caz în care se returneaază această unică valoare).

1. Se împart cele n elemente ale tabloului de intrare în $\lfloor \frac{n}{5} \rfloor$ grupe de câte 5 elemente fiecare şi cel mult un grup să conţină restul de $n \mod 5$ elemente.

2. Se determină mediana fiecăreia din cele $\lceil \frac{n}{5} \rfloor$ grupe cu sortarea prin inserţie, ordonând elementele fiecărui grup (cu cel mult 5 elemente), apoi se aleg medianele din listele sortate, corespunzătoare grupelor. Dacă în ultimul grup avem număr par de elemente, se va considera cel mai mare din cele două mediane.

3. Se utilizează recursiv Selectie pentru a găsi mediana x din cele $\lceil \frac{n}{5} \rfloor$ mediane găsite la pasul precedent.

4. Se partiţionează tabloul în jurul medianei medianelor x, utilizând o versiune modificată a algoritmului Partitie. Fie k elemente din prima parte a partiţionării.

5. Se utilizează recursiv selecţie pentru a găsi al i–lea cel mai mic element din partea inferioară (dacă $i \leq k$), sau al $(i-k)$–lea cel mai mic element din partea a doua a partiţiei (pentru $i > k$).

În [2] se arată că se ajunge la o complexitate $T(n) = O(n)$.

9.6 Heapsort

Fie vectorul:

$v =$

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Arborele ataşat este cel din Figura 9.2 (a se revedea secţiunea 4.5.2, pagina 36). Acest arbore are adâncimea de $\lfloor \log_2 n \rfloor$.

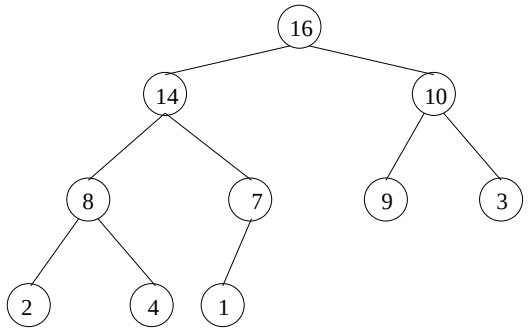


Figura 9.2: Arborele ataşat vectorului v

O structură căreia i se poate pune în corespondenţă un arbore binar echilibrat se numeşte HeapMax, dacă oricare nod are o valoare mai mare decât oricare din fiii săi. Dacă orice nod are valoarea mai mică decât oricare dintre fii, atunci structura se numeşte HeapMin.

Algoritmul de heapificare a unui vector cu n elemente începând de la a i-a componentă este:

Heapify(a, n, i)
$imax = i$
repetă
  $i = imax$
  $l = 2 * i$
  $r = l + 1$
  dacă $l \leq n$ şi $a(l) > a(i)$ *atunci*
    $imax = l$
  sfârşit dacă
  dacă $r \leq n$ şi $a(r) > a(imax)$ *atunci*
    $imax = r$
  sfârşit dacă
  dacă $i \neq imax$ *atunci*
    $a(i) \leftrightarrow a(imax)$

sfârşit dacă
până când $i = imax$
Return

Algoritmul pentru costrucţia unui heap este următorul:

ConstruiesteHeap(a, n)
pentru $i = [n/2], 1, -1$
Heapify(a, n, i)
sfârşit pentru
Return

Fie şirul:

$$a = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 4 & 1 & 3 & 2 & 16 & 9 & 10 & 14 & 8 & 7 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array}$$

Evoluţia arborelui la apelul ConstruiesteHeap$(a, 10)$ este arătată în Figura 9.6

Complexitatea: pentru un nod de înălţime h, complexitatea algoritmului Heapify este de $\Theta(h)$. Se poate arăta că pentru un heap cu n noduri există cel mult $\lfloor \frac{n}{2^{h+1}} \rfloor$ noduri de înălţime h. Deci timpul de execuţie pentru ConstruiesteHeap este:

$$T(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right)$$

Dar

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Deci timpul de execuţie pentru algoritmul ConstruiesteHeap este $O(n)$.

Folosim cei doi algoritmi de mai sus pentru sortare în modul următor:

HeapSort(a, n)
ConstruiesteHeap(a, n)
pentru $i = n, 2, -1$
$a(1) \leftrightarrow a(i)$
Heapify$(a, i-1, 1)$
sfârşit pentru
Return

Sortarea cu HeapSort se face cu complexitatea $O(n) + nO(\log_2 n) = O(n \log_2 n)$. Deoarece sortatea folosind comparaţii are complexitatea inferioară $\Omega(n \log_2 n)$ (secţiunea 8.4), rezultă că algoritmul Heapsort are complexitatea $\Theta(n \log n)$.

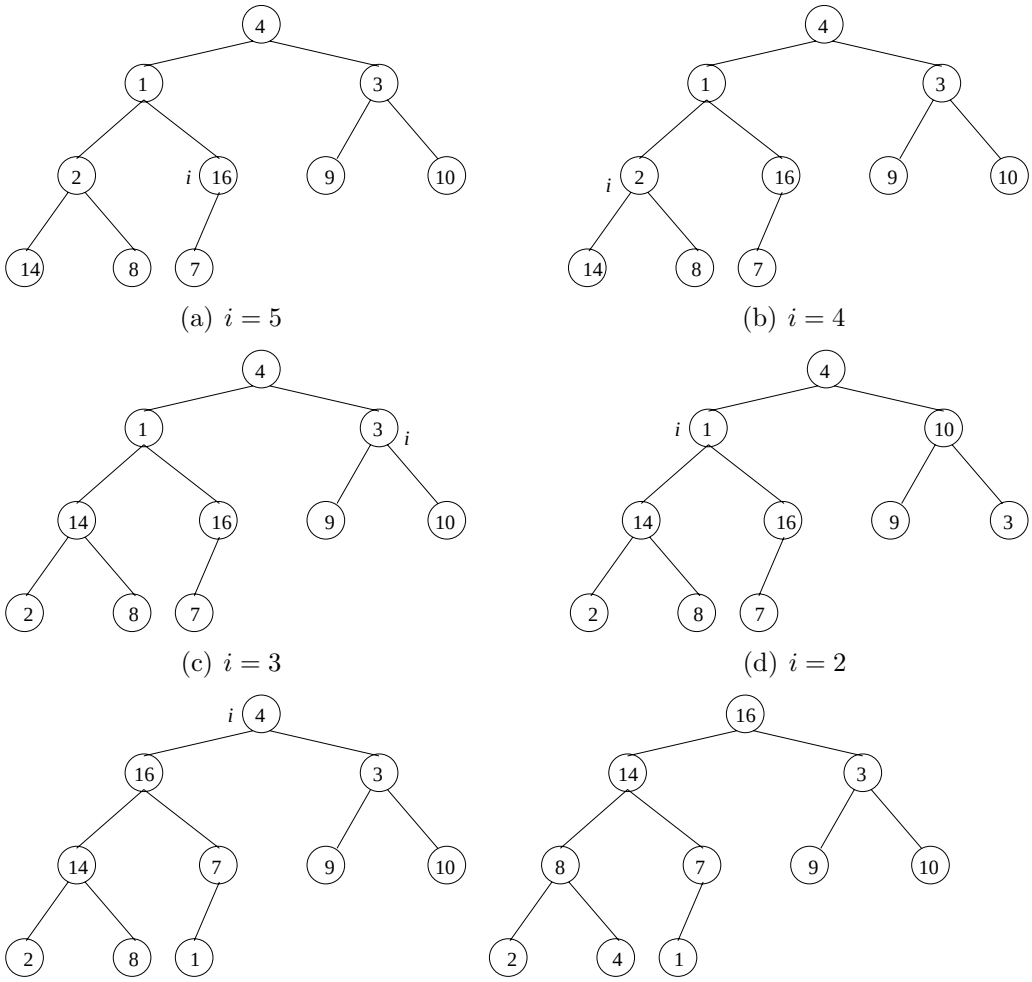
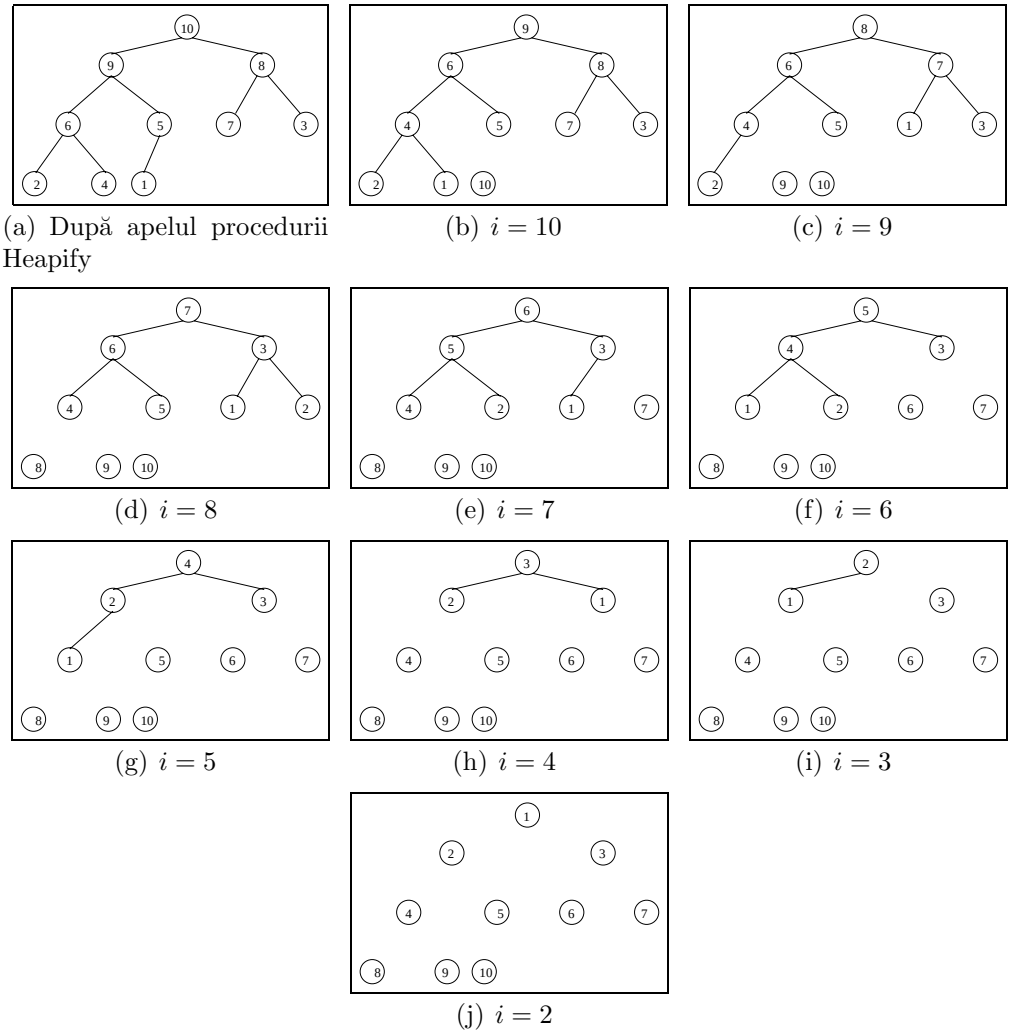


Figura 9.3: Obţinerea unui maxheap din şirul $(4, 1, 3, 2, 16, 9, 10, 14, 8, 7)$.

Exemplu: fie şirul: $a = (4, 1, 3, 2, 10, 7, 8, 9, 6, 5)$. Dupa apelul procedurii ConstruiesteHeap$(a, 10)$ se obţine heap-ul din Figura 9.5(a). Pentru apelurile Heapify$(a, i-1, 1)$ $(i = n, n-1, \ldots, 2)$ evoluţia este dată în Figurile 9.5(b)-9.5(j) (s–au pus în evidenţă prin detaşare elementele care nu vor mai face parte din heap).



(a) După apelul procedurii Heapify (b) $i = 10$ (c) $i = 9$

(d) $i = 8$ (e) $i = 7$ (f) $i = 6$

(g) $i = 5$ (h) $i = 4$ (i) $i = 3$

(j) $i = 2$

Figura 9.5: Evoluţia şirului şi a heap–ului în subalgoritmul Heapsort.

Rezultatul sortării şirului a este: $a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$.

Heapsort este un excelent algoritm de sortare, dar o implementare bună a QuickSort–ului duce la sortări mai rapide. Heap–ul este însă foarte util în ţinerea în evidenţă dinamică a cozilor de priorităţi unde o complexitate de n care s–ar putea obţine printr–un algoritm cu inserţie este înlocuită cu o complexitate de $\log n$ păstrând coada într–un heap.

9.7 Algoritmi de sortare în timp liniar

Există algoritmi de sortare a căror complexitate este mai mică decât cea teoretică de $\Omega(n \log_2 n)$, dar pentru situaţii speciale ale şirului. Astfel, dacă şirul este constituit din numere întregi pozitive (sau măcar întregi), sau dacă ele sunt numere distribuite uniform într–un interval, sau luînd în considerare cifrele numerelor din şir se pot da diferiţi algoritmi de complexitate liniară pentru sortare.

9.7.1 Algoritmul sortării prin numărare (CountSort)

Un astfel de algoritm este algoritmul de sortare prin numărare numit CountSort. Fie $(x(i))_{i=1,n}$ un şir de numere naturale strict pozitive şi k maximul acestui şir. Sortarea prin numărare constă în găsirea a câte elemente sunt mai mici decât $x(i)$ (pentru fiecare i); în acest fel vom şti pe ce poziţie trebuie să se afle $x(i)$ în şirul sortat (dacă sunt m elemente mai mici decât $x(i)$, atunci el ar trebui să se afle pe poziţia $m+1$ în şirul sortat).

Considerăm şirul iniţial $x = (x(1), x(2), \ldots, x(n))$; $y = (y(1), y(2), \ldots, y(n))$ va fi şirul sortat, iar $c = (c(1), c(2), \ldots c(k))$ este un şir ajutător.

CountSort(x, n, y, k)
pentru $i = 1, k$
  $c(i) = 0$
sfârşit pentru
pentru $j = 1, n$
  $c(x(j)) = c(x(j)) + 1$
sfârşit pentru
pentru $i = 2, k$
  $c(i) = c(i) + c(i-1)$
sfârşit pentru
pentru $j = n, 1, -1$
  $y(c(x(j))) = x(j)$
  $c(x(j)) = c(x(j)) - 1$
sfârşit pentru
Return

Exemplu: $x = (3, 6, 4, 1, 3, 4, 1, 4)$, $k = 6$.

- elementele lui c sunt iniţializate cu 0:

$$c = (0, 0, 0, 0, 0, 0)$$

- pentru i de la 1 la k, pe poziţia i din c obţinem numărul de elemente

egale cu i în şirul x:

$$c = (2, 0, 2, 3, 0, 1)$$

- pentru $i = 2, k$, pe poziţia i a şirului c se obţine câte elemente mai mici sau egale cu i sunt în şirul x:

$$c = (2, 2, 4, 7, 7, 8)$$

- pentru $j = n, 1$, cu pasul -1, se află componentele şirului y sortat:

$$y = (1, 1, 3, 3, 4, 4, 4, 4, 6)$$

Complexitatea algoritmului CountSort este $\Theta(n + k)$. Dacă $k = O(n)$ (adică există o constantă c astfel încât pentru orice n suficient de mare, să avem $k < cn$), atunci complexitatea algoritmului de sortare prin numărare este $\Theta(n)$.

9.7.2 Sortare pe baza cifrelor

Să presupunem că avem un şir de numere naturale, având toate acelaşi număr d de cifre (dacă nu, atunci un număr se poate completa cu zerouri nesemnificative până când atinge numărul maxim de cifre din şir). Se poate face ordonarea şirului de numere în felul următor: se porneşte de la cifra cea mai puţin semnificativă (ultima cifră) şi se sortează elementele pe baza acestei informaţii, obţinându–se o altă ordine (un alt şir). Se porneşte de al acest ultim şir generat şi se sortează elementele după următoarea cifră mai puţin semnificativă (penultima cifră), pentru care se face sortarea elementelor; şi aşa mai departe, în final făcându–se sortarea după cea mai semnificativă cifră (prima cifră). Esenţial este ca dacă sortarea se face după cifra i (numerotarea cifrelor o considerăm de la stânga spre dreapta, de la cifra cea mai semnificativă la cea cea mai puţin semnificativă), atunci se pleacă de la ordinea obţinută anterior pentru cifra $i + 1$ $(1 \leq i < d)$; în al doilea rând, algoritmul care este folosit pentru sortarea cifrelor trebuie să aibe următoarea proprietate: dacă într–un şir $x = (x(1), \ldots, x(n))$ avem că $x(i) = x(j)$ şi $i < j$, prin sortare se ajunge ca $x(i)$ să fie repartizat pe poziţia i', $x(j)$ să fie repartizat pe poziţia j', iar $i' < j'$ (spunem că *algoritmul de sortare este stabil*).

În pseudocod, algoritmul este:

OrdonarePeBazaCifrelor(a, d)
pentru $i = d, 1, -1$
  sortează stabil tabloul a după cifra i
sfârşit pentru

Return

Analiza algoritmului depinde în mod evident de complexitatea algoritmului prin care se face sortarea pe baza unei cifre. De exemplu, dacă numerele sunt în baza b, se poate folosi sortarea prin numărare din secţiunea precedentă (care este stabil). Pentru fiecare cifră după care se face sortarea avem timpul $\Theta(n+b)$, iar pentru sortarea completă (ciclul *pentru*) complexitatea este $\Theta(d(n+b))$. Dacă d este constant şi $b = O(n)$, atunci complexitatea este liniară.

9.7.3 Sortare pe grupe

Pentru algoritmul prezentat aici, în ipoteza în care numerele sunt uniform distribuite (dacă considerăm intervalul dat de minimul şi maximul din şir şi dacă împărţim acest interval în k subintervale de lungime egală, atunci şansa ca un număr oarecare din şir să se găsească în oricare din cele k subintervale este aceeaşi; altfel spus, nu exsite zone în care numerele să se concentreze în număr mult mai mare decât în alte regiuni de aceeaşi dimensiune), se poate da un algoritm care *în medie* are complexitatea $\Theta(n)$.

Vom considera fără a restrânge generalitatea că numerele sunt în intervalul $[0,1)$ (dacă nu sunt, atunci putem proceda astfel: determinăm $min = \min x(i), i = 1,n$ şi $max = \max x(i), i = 1,n$; se efectuează $x(i) = x(i) - min$, pentru $i = 1,n$ - în acest fel se aduc numerele în intervalul $[0, max - min]$; apoi se face împărţirea $x(i) = x(i)/(max - min + 1)$, prin care se aduc numerele la intervalul $[0,1)$; după sortare se fac operaţiile inverse, în ordinea înmulţire, adunare). principiul algoritmului constă în a împărţi intervalul $[0,1]$ în n intervale egale şi să introducem cele n numere în cele n grupe. Dacă numerele sunt uniform distribuite, nu ar trebui să apară diferenţe mari între grupe, din punct de vedere a câte numere conţin. Se sortează numerele din fiecare grupă, după care se trece prin fiecare grupă şi se obţine ordinea crescătoare.

Algoritmul de mai jos consideră că fiecare grupă este implementată ca o listă înlănţuită. Pentru subintervalul $(k \cdot \frac{1}{n}, (k+1) \cdot \frac{1}{n}), 0 \leq k < n$ vom avea lista înlănţuită $b(k)$.

SortarePeGrupe(a,n)
pentru $i = 1,n$
  inserează $a(i)$ în lista $b(\lfloor n \cdot a(i) \rfloor)$
sfârşit pentru
pentru $i = 0,n-1$
  Sortează lista $b(i)$ folosind sortarea prin inserţie
sfârşit pentru

Concatenează listele $b(0), b(1), \ldots, b(n)$
Return

Plecând de la proprietatea că numerele sunt uniform distribuite, se arată în [2] că timpul mediu pentru sortarea pe grupe este $O(n)$.

9.8 Rezumat

Noţiunea de statistică de ordine generalizează pe cele de minim, maxim, mediană. Selectarea celei de a i-a statistici de ordine se pate realiza în timp liniar. Pentru cazuri particulare ale vectorilor care se sortează, se pot folosi algoritmi liniari (se depăşeste deci bariera de $\Omega(n \log n)$ demonstrată pentru algoritmi care lucrează exclusiv pe baza comparaţiilor).

9.9 Exerciţii (rezolvările la pagina 85)

1. Descrieţi un algoritm care, fiind dată o mulţime S de n numere întregi şi distincte şi un întreg pozitiv $k \leq n$, determină cele k numere care sunt cele mai apropiate de mediana lui S.

2. Fie $a[1 \ldots n]$, $b[1 \ldots n]$ două şiruri sortate. Scrieţi un algoritm performant care găseşte mediana celor $2n$ numere.

3. Fiind dată o mulţime de n numere, dorim să găsim cele mai mari i numere în ordinea sortată, folosind un algoritm bazat pe comparaţii. Să se scrie mai multe variante şi să se compare din punct de vedere al complexităţii.

4. Să se modifice algoritmul CountSort astfel încât să sorteze şi şiruri conţinând numere întregi negative.

Capitolul 10

Metoda *Greedy*

10.1 Obiective

Capitolul va introduce o nouă metodă de rezolvare a problemelor. Ea va arăta cum se abordează problemele pentru care soluţia sse poate construi prin alegeri ale componentelor, alegeri asupra cărora nu se revine. Se doreşte a se exemplifica demonstrarea corectitudinii algoritmilor (fără de care acesta ar fi doar o euristică). Probleme clasice, ale căror rezultate se pot folosi în cazuri mai generale sunt enunţate si rezolvate (sau sunt făcute trimiteri bibliografice).

10.2 Prezentare generală

Metoda Greedy este o metodă de elaborare a algoritmilor ce determină $x^* \in X$ ce verifică o condiţie dată C. Exemplul semnificativ este cel al problemelor de optimizare în care se determină x^* ce minimizează sau maximizează o funcţie obiectiv. Să presupunem că soluţia x^* are mai multe componente şi că spaţiul soluţiilor X este cunoscut. De asemenea presupunem că C oferă un criteriu de selecţie prin care se pot alege valori pentru componentele soluţiei.

Modul de rezolvare al problemelor dat de strategia Greedy este acela de construcţie a soluţiei componentă cu componentă. Se pleacă de la soluţia vidă şi la fiecare pas al algoritmului se selectează o valoare pentru o componentă; dacă această valoare convine, atunci aceasta este pusă în soluţie pe componenta respectivă. Într–un cuvânt, *se ia tot ce este bun* pentru construcţia soluţiei. Această selecţie se face în funcţie de C, valoarea aleasă asigurând la pasul respectiv cea mai bună continuare în spiritul lui C.

Strategia descrisă anterior are un posibil inconvenient dat de modul de

selecţie. Dacă la fiecare pas alegem cea mai bună valoare în spiritul lui C, nu este necesar ca în final să obţinem soluţia optimă căutată. Este ceea ce se întâmplă în problemele de optimizare: o succesiune de creşteri poate să ducă la un optim local şi nu la cel global căutat. Dacă se reuşeste să se demonstreze că soluţia găsită verifică condiţia C atunci am obţinut o soluţie pentru problemă. În caz contrar, dacă soluţia găsită este suficient de aproape de soluţia optimă spunem că avem o *soluţie Greedy eurisitcă*.

Cele prezentate anterior pot fi descrise de procedura Greedy de mai jos:

Greedy(x)
$x = \emptyset$
cât timp soluţia nu este completă
  selectează o valoare a
  dacă valoarea a convine *atunci*
    $x = x \cup \{a\}$
  sfârşit dacă
sfârşit cât timp
Return

În concluzie o problemă se rezolvă prim metoda Greedy respectând etapele:

1. În funcţie de condiţia C se găseşte un criteriu de selecţie a valorilor
2. Se elaborează algoritmul respectând schema anterioară
3. Se demonstrează corectitudinea alegerii făcute arătând că soluţia x^* respectă C. În caz contrar se încearcă să se determine cât de departe este soluţia găsită de algoritm faţă de soluţia căutată.

Complexitate soluţiei Greedy este dată de complexitatea selecţiei valorii la fiecare pas şi de numărul de selecţii făcut de către algoritm. În general, metodele de tip Greedy sunt metode de complexitate polinomială mică, ceea ce este o justificare pentru utilitatea metodei.

Exemplifcăm metoda anterioară prin câteva probleme clasice.

10.3 Submulţimi de sumă maximă

Enunţul problemei: Dacă $x = (x(i), i = 1, n)$ reprezintă o mulţime de elemente atunci să se determine submulţimea $y = (y(i), i = 1, k)$ de sumă maximă.

Date de intrare: n - dimensiunea vectorului; $x = (x(i), i = 1, n)$, mulţimea de elemente;

Date de ieşire: k - numărul de elemente al submulţimii $y = (y(i), i = 1, k)$ - submulţimea de elemente.

Problema anterioară este o problemă clasică de optimizare care se rezolvă prin Greedy. Submulţimea căutată este o soluţie a problemei de optimizare

$$\max\left\{\sum_{y\in Y} y | \emptyset \neq Y \subseteq X\right\}$$

Simplitatea problemei de optimizare oferă un criteriu de seleţie foarte simplu bazat pe observaţiile:

1. dacă în mulţimea x sunt numere pozitive, adăugarea unuia la submulţimea y conduce la creşterea sumei; găsim astfel submulţimea y formată din toate elementele pozitive;

2. în cazul în care toate elementele sunt negative, submulţimea y este formată cu cel mai mare element.

Ca atare, la fiecare pas alegem în y elementele pozitive. Dacă nu sunt, atunci alegem în y elementul maximal.

SumaMax(n, x, k, y)
$k = 0$
pentru $i = 1, n$
  dacă $x(i) \geq 0$ *atunci*
    $k = k + 1$
    $y(k) = x(i)$
  sfârşit dacă
sfârşit pentru
dacă $k = 0$ *atunci*
  $max = x(1)$
  pentru $i = 2, n$
    dacă $x(i) > max$ *atunci*
      $max = x(i)$
    sfârşit dacă
  sfârşit pentru
  $k = 1$
  $y(k) = max$
sfârşit dacă
Return

Se observă că fiecare din cele n selecţii are complexitate $\Theta(1)$, găsind în final o complexitate pentru algoritm de $\Theta(n)$.

Teorema 2 (Corectitudine). *Submulţimea y determinată de către algoritm are suma elementelor maximă faţă de toate submulţimile lui x.*

Demonstraţie. Notăm $sum(x)$ suma elementelor din x. Fie z submulţimea de sumă maximă. Vom arăta că $y = z$ tratând cele două cazuri.

1. În x nu sunt elemente pozitive. Atunci $sum(z) \leq z(1) \leq \max\{x(i), i = 1, n\} = sum(y)$; din proprietatea lui z rezultă $sum(z) = sum(y)$ şi deci y este submulţime de sumă maximă.

2. În x sunt elemente pozitive. Atunci următoarele afirmaţii conduc la egalitatea $y = z$:

   (a) z nu conţine elemente negative. Dacă z conţine elementul a negativ, atunci $sum(z) = sum(z - \{a\}) + a < sum(z - \{a\})$ fals, pentru că z este de sumă maximală;

   (b) z conţine toate elementele pozitive. Dacă z nu conţine elementul strict pozitiv a atunci $sum(z) < sum(z + \{a\})$ fals.

   Rămâne că z nu conţine elemente negative şi conţine toate pozitivele, deci $y = z$

□

10.4 Arborele Huffman

Enunţul problemei: dacă $S = \{s_1, s_2, \ldots, s_n\}$ este o mulţime de semnale şi $p = (p(i), i = 1, n)$ este vectorul de probabilitate asociat lui S, atunci să se determine codul de lugime optimă asociat lui S.

Date de intrare: n - numărul de semnale, $p = (p(i), i = 1, n)$ - vectorul de probabilităţi.

Date de ieşire: codul optim reprezentat sub formă arborescentă.

Un exemplu de cod care simulează optimalitatea este alfabetul Morse. O caracteristică a acestui cod este aceea că simbolurile cele mai probabile sunt codificate cu mai puţine caractere, astfel încât costul transmisiei să fie cât mai mic. Înainte de a da rezolvarea problemei, este bine să facem o prezentare a câtorva noţiuni de teoria codurilor.

Un cod binar este o aplicaţie injectivă $\phi : S \to \{0, 1\}^+$. Lungimea medie a codului în funcţie de sistemul de probabilităţi este definită de $L(\phi) = \sum_{i=1}^{n} |\phi(s_i)|p(i)$ ($|a|$ este numărul de caractere a lui a). Un cod este optim dacă minimizează lungimea medie. Dacă $\phi : S \to \{0, 1\}^+$ este un cod instantaneu, atunci arborele m-ar asociat se notează cu T_ϕ şi verifică:

- are atâtea frunze câte semnale are S;

- drumul în arbore de la rădăcină la frunză $s \in S$ dă codificarea $\phi(s)$ prin concatenarea marcajelor muchiilor.

Lungimea medie a unui cod poate fi transportată într–o lungime medie a arborelui asociat, dată de $L(\phi) = \sum_{i=1}^{n} |\phi(s_i)|p(i) = \sum_{i=1}^{n} niv_{T_\phi}(s_i)p(i) = L(T_\phi)$.

Numim arbore Huffman asociat sistemului de numere reale $p = (p(i), i = 1, n)$ arborele ce minimizează $L(T) = \sum_{i=1}^{n} niv_T(s_i)p(i)$.

Vom lucra cu structuri arborescente reprezentate înlănţuit în care în câmpul de informaţie avem probabilitatea sau frecvenţa simbolului.

Până în acest moment nu am spus nimic despre modul de aplicare al stretegiei Greedy. Rămâne afirmaţia făcută iniţial că semnalele cele mai probabile sunt codificate cu cât mai puţine caractere.

Algoritmul lui Huffman construieşte codul optim prelucrând o *pădure* după regulile:

- iniţial pădurea este formată din vârfuri izolate reprezentând semnalele.

- se repetă o execuţie dată de:

  1. se selectează arborii pădurii având informaţia cea mai mică;
  2. se conectează cei doi arbori aleşi într–unul singur printr–o rădăcină comună având informaţia suma informaţiilor din arbori

Proceura Huffman determină arbore minim.

Huffman(n, s, p, rad)
pentru $i = 1, n$
  $s(i) \Leftarrow LIBERE$ {LIBERE este zona din care se fac alocări de memorie}
  $LS(s(i)) = *$
  $LD(s(i)) = *$
  $INFO(s(i)) = p(i)$
sfârşit pentru
Sort(n, s)
pentru $i = n, 2, -1$
  $x \Leftarrow LIBERE$
  $LS(x) = s(i)$
  $LD(x) = s(i-1)$
  $INFO(x) = p(i) + p(i-1)$
  $j = i - 1$
  pentru $j > 0$ şi $INFO(x) > INFO(s(j))$

$s(j+1) = s(j)$
$j = j - 1$
sfârşit pentru
$s(j+1) = x$
sfârşit pentru
$rad = s(1)$
Return

În algoritmul precedent procedura *Sort* determină o sortare a semnalelor $s(i), i = 1, n$ descrescător după $p(i)$.

Teorema 3 (Complexitate). *Algoritmul Huffman are complexitatea* $\Theta(n^2)$.

Demonstraţie. Complexitatea algoritmului este dată de complexitatea sortării şi de complexitatea seleţiilor repetate. Complexitatea sortăriiu este $\Theta(n \log n)$. Selecţiile repetate ce se fac utilizează principiul inserării directe. Pentru a insera semnalul p în structura $(s(j), j = 1, i-1)$ facem cel mult $i-1$ comparaţii, găsind în final un număr de $\sum_{i=1}^{n} i - 1 = n(n-1)/2 = \Theta(n^2)$ comparaţii. Rezultă complexitatea din enunţ. □

Teorema 4 (Corectitudine). *Algoritmul Huffman generează un cod optim în raport cu probabilităţile* $p = (p(i), i = 1, n)$.

Demonstraţie. Vom folosi în demonstraţie următoarele notaţii:

- $H = H(p_1, \ldots, p_n)$ - arborele dat de algoritmul lui Huffman
- $T = T(p_1, \ldots, p_n)$ - arborele optim, unde $p(i) = p_i, \forall i = 1, n$.

Despre arborele optim se poate arăta:

- $p_i < p_j$ rezultă $niv_T(s_i) > niv_T(s_j)$
- există un arbore optim în care cele mai puţin probabile semnale sunt fraţi.

Să presupunem că s_i şi s_j sunt semnalele cu probbabilitatea cea mai mică şi că prin unirea celor două semnale se creează un nou semnal notat $s_{i,j}$.

Algoritmul Huffman construieşte arborele binar după regula recursivă dată de figura 10.1. Avem următorul calcul de lungime:

$$L(H) = \sum_{i=1}^{n} niv(s_i) = \sum_{i=1}^{n-2} niv(s_i)p(i) + niv(s_{n-1})p(n-1) + niv(s_n)p(n)$$

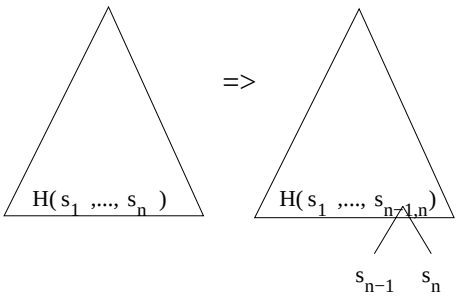


Figura 10.1: Crearea unui nod pe baza semnalelor cele mai puţin probabile

şi deoarece $niv(s_{n-1}) = niv(s_n)$ rezultă că:

$$\begin{aligned} L(H) &= \sum_{i=1}^{n-2} niv(s_i)p(i) + (p(n-1)+p(n))niv(s_{n-1}) - 1 + \\ &+ p(n-1)+p(n) = \\ &= \sum_{i=1}^{n-2} niv(s_i))p(i) + p(n-1,n)niv(s_{n,n-1}) + p(n-1) + p(n) \end{aligned}$$

găsind în final $L(H(p(1),\ldots,p(n))) = L(H(p(1),\ldots,p(n-1)+p(n)))+p(n-1)+p(n)$

Printr–un calcul analog se arată că şi arborele optim ce are pe $s_{n=1}$ şi s_n verifică:

$$L(T(p(1),\ldots,p(n))) = L(T(p(1),\ldots,p(n-1)+p(n))) + p(n-1) + p(n).$$

Demonstraţia optimalităţii se va face prin inducţie după n.

Pentru $n = 2$ algoritmul ne oferă $H(p(1), p(2))$ care codifică pe s_1 cu 0 şi pe s_2 cu 1, ceea ce înseamnă că este un cod optim.

Să presupunem că avem proprietatea adevărată pentru n semnale şi anume codul Huffman asociat semnalelor $s_1,\ldots,s_n$ este optim.

Fie $p = (p(i), i = 1, n+1)$ un sistem cu $n+1$ probabilităţi în care $p(n-1)$ şi $p(n-2)$ sunt cele mai mici două numere din şirul p. Aplicând formulele găsite anterior obţinem:

$$\begin{aligned} L(H(p(1),\ldots,p(n),p(n+1))) &= L(H(p(1),\ldots,p(n)+p(n+1)) + p(n) + p(n+1)) \\ L(T(p(1),\ldots,p(n),p(n+1))) &= L(T(p(1),\ldots,p(n)+p(n+1)) + p(n) + p(n+1)) \end{aligned}$$

Pentru că arborele Huffman codifică n semnale optim vom găsi:

$$L(T(p(1),\ldots,p(n)+p(n+1))) \geq L(H(p(1),\ldots,p(n)+p(n+1)))$$

Din relaţiile de mai sus vom găsi:

$$L(T(p(1),\ldots,p(n)+p(n+1)))+p(n)+p(n+1) \geq L(H(p(1),\ldots,p(n)+p(n+1)))+p(n)+p(n+1)$$

şi deci $L(T(p(1),\ldots,p(n),p(n+1))) \geq L(H(p(1),\ldots,p(n),p(n+1)))$ Din optimalitatea lui T vom găsi:

$$L(T(p(1),\ldots,p(n),p(n+1))) = L(H(p(1),\ldots,p(n),p(n+1)))$$

deci H este un arbore optim. □

10.5 Interclasare optimală

Enunţul problemei: Dându–se o mulţime de vectori sortaţi crescător să se elaboreze un algoritm care determină interclasarea acestora cu numărul minim de comparaţii.

Date de intrare: n - numărul de vectori, $nr = (nr(i), i = 1, n)$ - numărul de elemente din fiecare vector, $x(i) = (x(i,j), j = 1, nr(i))$ şirul de vectori. Date de ieşire: $y = (y(i), i = 1, nr(1) + \cdots + nr(n))$, vectorul interclasat.

Să presupunem că interclasarea a doi vectori se face prin algoritmul de interclasare directă. Dacă dimensiunile vectorilor sunt m, respectiv n atunci numărul de comparaţii este cel mult $m + n - 1$. În general această interclasare nu este una optimă dar facem presupunerea că orice doi vectori se interclasează folosind interclasarea directă. În funcţie de modul de interclasare al vectorilor de interclasare se obţin diferite soluţii cu număr distinct de comparaţii. Este deci esenţial să găsim o ordine care conduce către numărul minim de comparaţii.

Vom nota $x+y$ vectorul obţinut prin interclasarea lui x cu y. Dacă avem 4 vectori x, y, z, t cu respectiv 10, 20, 30 şi 40 de elemente atunci:

- interclasarea $((x+y)+z)+t$ se face cu (10+20-1)+(30+30-1)+(60+40-1)=187 comparaţii
- inteclasarea $((z+t)+y)+x$ se face cu (30+40-1)+(70+20-1)+(90+10-1)=257 comparaţii

deci prima strategie este mai bună decăt a doua (de fapt, este chiar cea optimă). Din acest exemplu se observă că maximă 40 se regăseşte în toate interclasările ce se efectuează conducând al un număr mare de comparaţii. Idea ar fi ca vectorii cu dimensiuni mici să se se interclaseze prima dată astfel

încât aceste valori mici să apară în sumele de interclasare. Deci la fiecare pas vom selecta pentru interclasare vectorii de dimensiune minimă.

Pentru ca selecţia să se facă cât mai uşor pšstrăm vectorii în matricea x astfel încât vectorul nr să fie sortat descrescător. În acest caz se selectează ultimii 2 vectori din matricea x. După realizarea interclasării noul vector se inserează înmatricea x folosind diferite tehnici: inserţia directă sau inserţia binară. Se repetă procesul până când rămâne un singur vector.

InterOpt(n, nr, x, z)
Sortare(n, nr, x)
pentru $i = n-1, 1, -1$
  Inter$(nr(i+1), x(i+1), nr(i), x(i), y)$
  $n1 = nr(i)$
  $n2 = nr(i+1)$
  $j = i-1$
  cât timp $j > 0$ şi $n(j) > n1 + n2$
    pentru $k = 1, nr(j)$
      $x(j+1, k) = x(j, k)$
    sfârşit pentru
    $nr(j+1) = nr(j)$
    $j = j-1$
  sfârşit cât timp
  pentru $k = 1, n1 + n2$
    $x(j+1, k) = y(k)$
  sfârşit pentru
  $nr(j+1) = n1 + n2$
sfârşit pentru
Return

În procedura anterioară procedura *Sortare* determină rearanjarea elementelor (liniilor) matricii x descrescător în funcţie de vectorul nr iar *Inter* face interclasare directă a doi vectori.

Pentru a analiza corectitudinea algoritmului definim printr–o formulă recursivă arborele asociat unei strategii de interclasare:

1. Arborele asociat unui vector este format dintr–un nod ce are ca informaţie numărul de elemente ale vectorului;

2. arborele asociat interclasării a doi vectori $x(1)$ şi $x(2)$ este reprezentat grafic în figura 10.2;

3. Dacă T_1 este arborele asociat şirului de vectori $x(i_1), \ldots, x(i_k)$ iar T_2 este arborele asociat şirului de vectori $x(i_{k+1}), \ldots, x(i_n)$ atunci T din figura 10.3 este arborele asociat prin strategie şirului de vectori $x(i_1), \ldots, x(i_n)$.

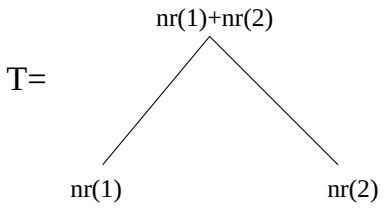


Figura 10.2: Arbore construit pe baza a doi vectori

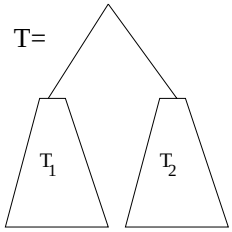


Figura 10.3: Arbore construit pe baza a două subsecvenţe de vectori.

Arborele asociat unui şir de vectori verifică:

- are atâtea frunze câţi vectori sunt în şirul de vectori;
- informaţia ataşată unui nod este suma dimensiunilor vectorilor din subarborii stâng şi drepţi ai nodului;
- dacă un vector apare pe nivelul p atunci termenii şirului apar în exact p interclasări;
- numărul total de comparaţii dat de strategie este $L(T) = \sum_{i=1}^{n} nr(i) \cdot niv(i) - n + 1$ unde $niv(i)$ este nivelul în T al vectorului $x(i)$

Teorema 5 (Corectitudine). *Algoritmul* InterOpt *determină interclasarea vectorilor* $x(i), i = 1, n$ *cu număr minim de comparaţii realizate din interclasările directe.*

Demonstraţie. Se observă că arborele asociat unei strategii de interclasare ete arborele Huffman asociat numerelor $nr(1), \ldots, , nr(n)$. Algoritmul anterior repetă modul de construcţie al arborelui Huffman deci suma $\sum_{i=1}^{n} nr(i) \cdot niv(i)$ este minimă şi implicit $L(T)$ este minim. Rezultă că strategia de interclasare este optimă. □

Teorema 6 (Complexitate). *Dacă L este numărul de comparaţii rezultat din interclasările directe, atunci algoritmul* InterOpt *are complexitatea* $\Theta(L+n^2)$.

Demonstraţie. Complexitatea algoritmului este dată de numărul de comparaţii făcut de interclasările directe şi de numărul de comparaţii dat de păstrarea structurii x. În primul caz numărul este n. Pentru cel de al doilea caz se observă că inserarea directă foloseşte cel mult $i-1$ comparaţii pentru ounerea şirului y în structura x. Găsim un număr total de $\sum_{i=2}^{n}(i-1) = \Theta(n^2)$ comparaţii.

Dacă se utilizează o inserare binară în structura x atunci complexitatea algoritmului devine $\Theta(L + n \log n)$. □

10.6 Rezumat

Metoda Greedy înseamnă construirea pas cu pas a soluţiei, prin alegerea componentelor pe baza unei strategii deduse din enunţul problemei. Spre deosebire de metoda Backtracking, nu se revine asupra unei decizii. Ea poate fi folosită şi pentru elaborarea unor algoritmi euristici (care nu determină optimul, sau pentru care o demonstraţie de corectitudine nu este dată). Pentru fiecare rezolvare greedy trebuie să se încerce justificarea prin metode matematice a optimalităţii soluţiei.

10.7 Exerciţii (rezolvările la pagina 99)

1. Dacă $x = (x(i), i = 1, m)$, $y = (y(i), i = 1, n)$ reprezintă două mulţimi de elemente, atunci să se determine mulţimea intersecţie $z = (z(i), i = 1, k)$.

2. Dându–se n obiecte de cost $c = (c(i), i = 1, n)$ cu greutatea $g = (g(i), i = 1, n)$ şi un rucsac de capacitate maximă G, să se elaboreze un algoritm de umplere a rucsacului de cost maxim. Un obiect poate fi pus în rucsac într–un anumit procent (problema *continuă* a rucsacului).

3. Într–o sală, într–o zi trebuie planificate n spectacole. Pentru fiecare spectacol se cunoaşte intervalul în care se desfăşoară: $[st, sf)$. Se cere să se planifice un număr maxim de spectacole astfel încât să nu se suprapună..

4. Se dau n numere întregi nenule $b_1, \ldots, b_n$ şi m numere întregi nenule $a_1, \ldots a_m$, $n \geq m$. Să se determine o submulţime a mulţimii $B = \{b_1, \ldots, b_n\}$ care să maximizeze valoarea expresiei:

$$E = a_1x_1 + a_2x_2 + \cdots + a_mx_m$$

unde $x_i \in B$..

5. O staţie de servire trebuie să satisfacă cererile a n clienţi. Timpul de servire pentru fiecare client este cunoscut: pentru clientul i timpul este t_i. Să se minimizeze timpul total de aşteptare

$$T = \sum_{i=1}^{n} (\text{timpul de aşteptare pentru clientul } i)$$

Capitolul 11

Programare dinamică

11.1 Obiective

În acest capitol se prezintă metoda programării dinamice. După parcurgerea materialului, cititorul al trebui să fie familiarizat cu trăsăturile generale ale unei probleme pentru care se apelează la programare dinamică şi să îşi însuşească strategia generală de rezolvare a unei probleme folosind această metodă.

11.2 Prezentare generală

Programarea dinamică, asemenea metodelor "*Divide et Impera*", rezolvă problemele combinând soluţiile unor subprobleme. "Programarea" în acest context se referă la o metodă tabelară şi nu la scrierea unui cod pentru calculator. Am văzut, în capitolul referitor la *divide et impera*, că algoritmii construiţi cu această metodă partiţionează problema în subprobleme indepenedente pe care le rezolvă în general recursiv, după care combină soluţiile lor pentru a obţine soluţia problemei iniţiale. Spre deosebire de această abordare, programarea dinamică este aplicabilă atunci când aceste subprobleme nu sunt independente, adică au în comun subprobleme[1].

În general, programarea dinamică se aplică problemelor de optimizare. Fiecare soluţie are o valoare şi se doreşte determinarea soluţiei optime (minim sau maxim). O asemenea soluţie numeşte soluţie optimă a problemei, prin contrast cu valoarea optimă, deoarece pot exista mai multe soluţii care să realizeze valoarea optimă.

[1]Termenul "dinamică" în acest conext nu are nimic în comun cu *alocarea dinamică*, pomenită în capitolul 4.

Dezvoltarea unui algoritm bazat pe metoda programării dinamice poate fi împărţită într–o secvenţă de patru paşi:

1. Caracterizarea structurii unei soluţii optime
2. Definirea recursivă a valorii unei soluţii optime
3. Calculul valorii unei soluţii optime într–o manieră "bottom–up" (plecând de la subprobleme de dimensiune mică şi ajungând la unele de dimensiuni din ce în ce mai mari)
4. Construirea unei soluţii optime din informaţia calculată.

Paşii 1–3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se doreşte doar calculul valorii optime. În vederea realizării pasului 4, deseori se păstrează informaţie suplimentară de la execuţia pasului 3, pentru a uşura construcţia unei soluţii optimale.

11.3 Înmulţirea unui şir de matrici

Primul nostru exemplu de programare dinamică este un algoritm care rezolvă problema înmulţirii unui şir de matrici. Se dă un şir $A_1, A_2, \ldots, A_n$ de n matrice care trebuie înmulţite. Acest produs se poate evalua folosind algoritmul clasic de înmulţire a unei perechi de matrici ca subalgoritm, o dată ce produsul $A_1 \cdot \cdots \cdot A_n$ este parantezat (această restricţie este impusă de faptul că înmulţire matricilor este operaţie binară). Un produs de matrici este complet parantezat dacă este format dintr–o sigură matrice sau dacă este format din factori compleţi parantezaţi.

Exemplu: produsul $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ poate fi complet parantezat în 5 moduri distincte, astfel:

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$
$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$
$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$
$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$
$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

Modul în care parantezăm un şir de matrici poate avea un impact dramatic asupra costului evaluării produsului. Să considerăm mai întâi costul înmulţirii a două matrici. Algoritmul standard este dat de următoarea procedură, descrisă în pseudocod. Prin *linii* şi *coloane* înţelegem numărul de linii şi coloane ale matricilor implicate.

Inmultire(A, n, m, B, p, q, C)

```
dacă m ≠ p atunci
   scrie( "dimensiuni incompatibile!" )
altfel
   pentru i = 1, n
      pentru j = 1, q
         C(i, j) = 0
         pentru k = 1, m
            C(i, j) = C(i, j) + A(i, k) · B(k, j)
         sfârşit pentru
      sfârşit pentru
   sfârşit pentru
sfârşit dacă
Return
```

Două matrici se pot înmulţi numai dacă numărul de coloane ale primei matrici este egal cu numărul de linii ale celei de a doua matrici, adică dacă avem $A(n, m)$ şi $B(m, q)$; în acest caz $C = AB$ va avea n linii şi q coloane. Timpul necesar pentru calculul matricei C este dat de numărul de înmulţiri scalare, $n \cdot m \cdot q$.

Pentru a ilustra modul în care apar costuri diferite la parantezări diferite ale produsului de matrici, să considerăm problema şirului A_1, A_2, A_3. Să presupunem că dimensiunile matricilor sunt 10×100, 100×5, 5×50. Dacă efectuăm înmulţirile pentru $((A_1 \cdot A_2) \cdot A_3)$, atunci vom avea $10 \times 100 \times 5 = 5000$ înmulţiri scalare pentru a efectua $(A_1 \cdot A_2)$, care va fi o matrice de dimensiune 10×5, plus alte $10 \times 5 \times 50 = 2500$ înmulţiri pentru a efectua $(A_1 \cdot A_2) \cdot A_3$, deci în total rezultă 7500 înmulţiri scalare. Pentru parantezarea $(A_1 \cdot (A_2 \cdot A_3))$ vom avea $100 \times 5 \times 50 = 25000$ înmulţiri scalare pentru a efectua $A_2 \cdot A_3$, care va fi o matrice de dimensiune 100×50, plus alte $10 \times 100 \times 50 = 50000$ înmulţiri scalare pentru a efectua $A_1 \cdot (A_2 \cdot A_3)$. Deci rezultă un număr de 75000 înmulţiri scalare.

Enunţul formalizat al problemei este: dându–se un şir $A_1, A_2, \ldots, A_n$ de n matrici, unde pentru $i = 1, n$ matricea A_i are dimensiunle $p_{i-1} \times p_i$, să parantezeze complet produsul $A_1 \cdot A_2 \cdot \cdots \cdot A_n$ astfel încât să se minimizeze numărul de înmulţiri scalare.

Înainte de a rezolva problema înmulţirii şirului de matrici prin programare dinamică, trebuie să ne asigurăm că verificarea completă a tuturor parantezărilor nu duce la un algoritm eficient. Fie $P(n)$ numărul de parantezări distincte ale unei secvenţe de n matrice. O secvenţă de n matrice o putem diviza între matricele k şi $k + 1$ pentru orice $k = 1, \ldots, n - 1$şi apoi putem descompune în paranteze, în mod independent, fiecare dintre secvenţe.

În acest fel vom obţine următoarea recurenţă:

$$P(n) = \begin{cases} 1 & \text{dacă } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{dacă } n \geq 2 \end{cases} \tag{11.1}$$

Numerele definte de (11.1) se numesc *numerele lui Catalan* şi se arată că $P(n) = \frac{1}{n}C_{2n-2}^{n-1}$, de unde $P(n) = \Omega(\frac{4^n}{n^2})$. Numărul de soluţii este exponenţial, deci ar fi o strategie slabă.

Primul pas în schema generală a metodei programării dinamice este dat de caracterizarea unei soluţii optimale. Pentru problema noastră descrierea este făcută în continuare. Vom face următoarea convenţie de notare: matricea obţinută în urma procesului de evaluare a produsului $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ se va nota cu $A_{i\ldots j}$. O parantezare optimă a produsului $A_1 \cdot \ldots \cdot A_n$ împarte produsul între A_k şi A_{k+1} pentru un anumit k din intervalul $1 \ldots n$. Aceasta înseamnă că pentru o valoare a lui k, mai întâi calculăm matricele $A_{1\ldots k}$ şi $A_{k+1\ldots n}$ şi apoi le înmulţim pentru a obţine rezultatul final $A_{1\ldots n}$. Costul acestei parantezări optime este dat de suma costurilor calculului pentru matricele $A_{1\ldots k}$, $A_{k+1\ldots n}$ şi $A_{1\ldots n}$. Observaţia care trebuie făcută este că: parantezarea subşirului prefix $A_1 \cdot \ldots \cdot A_k$ în cadrul parantezării optime a produslui $A_1 \cdot \ldots \cdot A_n$ este o parantezare optimă pentru $A_1 \cdot \ldots \cdot A_k$, pentru că altfel, dacă am presupune că mai există o metodă de parantezare mai puţin costisitoare a lui $A_1 \cdot \ldots \cdot A_k$ ar contrazice faptul că parantezarea pentru $A_1 \cdot \ldots \cdot A_n$ este optimă. Această observaţie este valabilă şi pentru parantezara lui $A_{k+1} \cdot \ldots A_n$. Prin urmare, o soluţie optimă a unei instanţe a unei probleme conţine soluţii optime pentru instanţe ale subproblemelor.

Al doilea pas în aplicarea metodei programării dinamice este dat de definirea valorii unei soluţii optime în mod recursiv, în funcţie de soluţiile optime ale subproblemelor. În cazul problemei înmulţirii şirului de matrice, o subproblemă constă în determinarea costului minim al unei parantezări a şirului $A_i \cdot A_{i+1} \cdot \cdots \cdot A_j$, pentru $1 \leq i \leq j \leq n$. Fie $m(i,j)$ numărul minim de înmulţiri scalare necesare pentru a calcula matricea $A_{i\ldots j}$; costul modalităţilor optime de calcul al lui $A_{1\ldots n}$ va fi $m(1,n)$. Definiţia recursivă a lui $m(i,j)$ este dată de formula (11.2)

$$m(i,j) = \begin{cases} 0 & \text{dacă } i = j \\ \min_{i \leq k < j} \{m(i,k) + m(k+1,j) + p_{i-1}p_kp_j\} & \text{dacă } i < j \end{cases} \tag{11.2}$$

Valorile $m(i,j)$ exprimă costul soluţiilor optime ale subproblemelor. Pentru a putea urmări modul de construcţie a slouţiei optime, să definim $s(i,j)$ care va conţine valoarea k pentru care împărţirea produsului $A_i \cdot \ldots A_j$ produce o parantezare optimă. Aceasta înseamnă că $s(i,j)$ este egal cu valoarea lui k pentru care $m(i,j) = m(i,k) + m(k+1,j) + p_{i-1}p_kp_j$.

În acest moment este uşor să scriem un algoritm recursiv dat de recurenţa (11.2) care să calculeze costul minim $m(1, n)$ pentru produsul $A_1 \cdot \ldots A_n$. Dar acest algoritm necesită timp exponenţial - nu mai bun decât căutarea completă a parantezărilor. Observaţia care se impune în acest moment se referă la faptul că avem relativ puţine subprobleme: o problemă pentru fiecare alegere a lui i şi j ce satisfac $1 \leq i \leq j \leq n$, adică un total de $C_n^2 + n = \Theta(n^2)$. Un algoritm recursiv poate întâlni fiecare subproblemă de mai multe ori pe ramuri diferite ale arborelui său de recurenţă. Această proprietate de suprapunere a subproblemelor este a doua caracteristică a programării dinamice.

În loc să calculăm recursiv soluţia recurenţei (11.2), vom aplica pasul al treilea din schema programării dinamice şi vom calcula costul optimal cu o abordare "bottom–up". Algoritmul următor presupune că matricele A_i au dimensiunile $p_{i-1} \times p_i$ pentru orice $i = 1, \ldots, n$. Intrarea este secvenţa $(p_0, \ldots, p_n)$ de $n+1$ elemente. Procedura foloşeste un tablou auxiliar $m(1 \ldots n, 1 \ldots n)$ pentru costurile $m(i, j)$ şi un tablou auxiliar $s(1 \ldots n, 1 \ldots n)$ care înregistrează acea valoare a lui k pentru care s–a obţinut costul optim în calculul lui $m(i, j)$, conform (11.2).

```
InmultireMatrici(p, m, s, n)
pentru i = 1, n
    m(i, i) = 0
sfârşit pentru
pentru l = 2, n
    pentru i = 1, n − l
        j = i + l − 1
        m(i, j) = ∞
        pentru k = i, j − 1
            q = m(i, k) + m(k + 1, j) + p_{i−1} p_k p_j
            dacă q < m(i, j) atunci
                m(i, j) = q
                s(i, j) = k
            sfârşit dacă
        sfârşit pentru
    sfârşit pentru
sfârşit pentru
Return
```

Algoritmul completează tabloul m într–un mod ce corespunde rezolvării problemei parantezării unor şiruri de matrici din ce în ce mai mari. Relaţia (11.2) arată că $m(i, j)$, costul de calcul al sirului de $j - i + 1$ matrici depinde doar de costurile calculării produselor şirurilor de mai puţin de $j - i + 1$ matrici. Aceasta însemnaă că, pentru $k = i, i + 1, \ldots, j - 1$, matricea $A_{i \ldots k}$ este un

produs de $k-i+1 < j-i+1$ matrice, iar matricea $A_{k+1\ldots j}$ este un produs de $j-k < j-i+1$ matrici.

În ciclul *pentru* $i = 1, n$ algoritmul iniţializează $m(i,i) = 0$ (costul minim al şirurilor de lungime 1). La prima execuţie a ciclului *pentru* $l = 2.n$ se calculează, cu formula (11.2), $m(i, i+1)$ pentru $i = 1, 2, \ldots, n-1$ (costul minim al şirurilor de lungime 2). La a doua trecere prin ciclul *pentru* $l = 2, n$ se calculează $m(i, i+2)$, pentru $i = 1, 2 \ldots, n-2$ (costul minim al şirurilor de lungime 3), etc. La fiecare pas, costul $m(i,j)$ calculat in ciclul *pentru* $k = i, j-1$ depinde doar de intrările $m(i,k)$ şi $m(k+1,j)$ ale tabloului, deja calculate.

În tabelul 11.2 de mai jos este descrisă funcţionarea algoritmului pentru un şir de $n = 6$ matrici având dimensiunile date în tabelul 11.1.

Matrice	Dimensiune
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Tabelul 11.1: Dimensiunile matricelor care se înmulţesc

1	2	3	4	5	6	1	2	3	4	5	6
0	15750	7875	9375	11875	15125	0	1	1	3	3	3
0	0	2625	4375	7125	10500	0	0	2	3	3	3
0	0	0	750	2500	5375	0	0	0	3	3	3
0	0	0	0	1000	3500	0	0	0	0	4	5
0	0	0	0	0	5000	0	0	0	0	0	5
0	0	0	0	0	0	0	0	0	0	0	0

Tabelul 11.2: Evoluţia algoritmului pentru matricile din tabelul 11.1. În partea stângă apare matricea m, în partea dreaptă matricea s.

În tablourile m, s sunt utilizate doar elementele de desupra diagonalei principale. Numărul minim de înmulţiri scalare necesare înmulţirii acestor şase matrici este $m(1,6) = 15125$. Pentru a calcula cu formula $q = m(i,k) +$

$m(k+1,j)+p_{i-1}p_kp_j$ pe $m(2,5)$, procedăm ca mai jos:

$$m(2,5)=\left\{\begin{array}{l} m(2,2)+m(3,5)+p_1p_2p_5=13000 \\ m(2,3)+m(4,5)+p_1p_3p_5=7125 \\ m(2,4)+m(5,5)+p_1p_4p_5=11375 \end{array}\right\}=7125$$

O simplă examinare a algoritmului conduce la constatarea că timpul de execuţie este $\Theta(n^3)$. Deşi acest algoritm determină numărul optim de înmulţiri scalare necesare pentru calculul produsului şirului de matrici, acesta nu prezintă în mod direct modul în care se face înmulţirea. Pasul 4 al schemei generale a metodei programării dinamice urmăreşte construirea unei soluţii optime din informaţia disponibilă.

În acest caz particular, vom folosi tabloul s pentru a determina modul optim de înmulţire a matricilor. Fiecare element $s(i,j)$ conţine valoarea lui k pentru care parantezarea optimă a produsului $A_i\cdot\ldots A_j$ împarte produsul între A_k şi A_{k+1}. Atunci ştim că în produsul final de calcul al matricel $A_{1\ldots n}$ optimul este $A_{1\ldots s(1,n)}\cdot A_{s(1,n)+1\ldots n}$. Înmulţirile anterioare pot fi determinate recursiv, deoarece $s(1,s(1,n))$ determină ultima înmulţire matriceală din calculul lui $A_{1\ldots s(1,n)}$; analog, $s(s(1,n)+1,n)$ determină ultima înmulţire din produsul $A_{s(1,n)+1,n}$.

11.4 Cel mai lung subşir crescător

Problema se enunţă în felul următor: se dă un şir de numere $a=(a(i),i=1,n)$. Să se determine cel mai lung subşir crescător. Prin subşir se înţelege un şir $a(i_1),a(i_2),\ldots,a(i_k)$, unde $1\le i_1<i_2<\cdots<i_k\le n$ (elementele subşirului nu sunt neapărat adiacente în şirul iniţial).

Dacă am considera toate subşirurile de elemente, atunci acestea ar fi în număr de 2^n-1. Chiar şi dacă am genera prin backatracking subşirurile crescătoare ale lui a, tot la complexitate exponenţială s–ar ajunge.

Ideea este de a calcula, pentru fiecare element al şirului a cel mai lung subşir crescător care începe cu elementul respectiv. În final, se alege cel mai mare subşir crescător din cele determinate pentru fiecare element. Să presupunem că cel mai lung subşir crescător care conţine elementul $a(p)$ este $a(i_1),\ldots,a(p),\ldots,a(i_k)$ (1). Facem afirmaţia că subşirul $a(p),\ldots,a(i_k)$ este cel mai lung subşir crescător care se formează începând cu $a(p)$. Dacă prin absurd nu ar fi aşa, atunci ar însemna că am avea un alt subşir crescător care să înceapă cu $a(p)$: $a(p),\ldots a(i_{k'})$(2). Ori, dacă am considera subşirul $a(i_1),\ldots,a(p),\ldots a(i_{k'})$ (partea finală a lui (1) este înlocuită cu subşirul (2)), atunci am avea un subşir crescător care să îl conţină pe $a(p)$ mai lung decât în cazul (1), ceea ce contrazice faptul că am presupus (1) maximal. Deci optimul

total implică optimul local, o caracteristică a problemelor de programare dinamică. Afirmaţiile de mai sus reprezintă suportul pentru strategia de rezolvare aleasă.

Vom considera un şir auxiliar $L = (L(i), i = 1, n)$, care va avea pe poziţia i lungimea celui mai lung subşir crescător care începe cu valoarea $a(i)$. Relaţia pe care o respectă elementele acestui şir este:

$$L(i) = \begin{cases} 1, & \text{dacă } i = n \\ 1 + \max\{L(j) | j > i, a(j) \geq a(i)\}, & \text{dacă } i < n \end{cases} \tag{11.3}$$

unde pentru cazul mulţimii vide de la a doua variantă maximul se ia 0. Valorile lui L se vor calcula iterativ (implementarea recursivă ar fi extrem de ineficientă, datorită calculului repetat care s–ar face pentru aceleaşi valori), în ordinea $L(n), L(n-1), \ldots, L(1)$. Pentru a determina mai uşor subşirul de lungime maximă, se construieşte un şir $sol = (sol(i), i = 1, n)$ care va conţine pentru valoarea i acel j care dă minimul pentru 11.3, sau chiar valoarea i dacă un asemenea j nu există. Determinarea lungimii maxime a unui subşir se face apoi calculând $\max_{i=1,n}(L(i))$, iar determinarea efectivă a acestui şir se face pe baza informaţiei din sol.

Exemplu: $a = (3, 2, 8, 6, 9, 7)$, $n = 6$. Atunci:

- $L(6) = 1$, $sol(6) = 6$
- $L(5) = 1 + \max \emptyset = 1$, $sol(5) = 5$
- $L(4) = 1 + \max\{L(5), L(6)\} = 2$, $sol(4) = 5$
- $L(3) = 1 + \max\{L(5)\} = 2$, $sol(3) = 5$
- $L(2) = 1 + \max\{L(3), L(4), L(5), L(6)\} = 3$, $sol(2) = 3$
- $L(1) = 1 + \max\{L(3), L(4), L(5), L(6)\} = 3$, $sol(1) = 3$

Valoarea maximă din vectorul L este 3, obţinută pentru $L(1)$ (şi altele). Pe baza vectorului sol obţinem şi şirul crescător maximal: 2, urmat de elementul de indice $sol(1) = 3$, adică $a(3) = 8$, care este urmat de elementul de indice $sol(3) = 5$, adică $a(5) = 9$, pentru care avem $sol(5) = 5$, adică nu are succesor.

```
Start SubsirCrescator
  citeşte( n, (a(i), i = 1, n) )
  pentru i = n, 1, −1
    L(i) = 1
    sol(i) = i
```

```
        pentru j = i + 1, n
            {Acest ciclu nu se execută niciodată pentru i = n}
            dacă a(j) ≥ a(i) şi 1 + L(j) > L(i) atunci
                L(i) = 1 + L(j)
                sol(i) = j
            sfârşit dacă
        sfârşit pentru
    sfârşit pentru
    LMax = L(1)
    pozLMax = 1
    pentru i = 2, n
        dacă L(i) > LMax atunci
            LMax = L(i)
            pozLMax = i
        sfârşit dacă
    sfârşit pentru
    scrie( "Lungimea maximă a unui subşir crescător este:", LMax )
    scrie( a(LMax) )
    i = pozLMax
    cât timp sol(i) ≠ i
        i = sol(i)
        scrie( a(i) )
    sfârşit cât timp
Stop
```

Complexitatea acestui algoritm este $\Theta(n^2)$, datorită celor două cicluri *pentru* imbricate. Menţinăm că nu este absolut necesar vectorul *sol*, dar determinarea soluţiei se face mai uşor pe baza lui.

11.5 Rezumat

Programarea dinamică este o metodă care se foloseşte în cazurile în care optimul general implică optimul parţial; în astfel de cazuri, demonstraţia se face de cele mai multe ori prin reducere la absurd. Pentru evitarea calculării aceloraşi rezultate, se apelează la tabelare – rezultatele pentru cazurile luate în considerare sunt memorate într–o structură de tip matricial. Pe baza acestor informaţii, se poate determina soluţia (soluţiile, chiar) pentru care se atinge optimul.

11.6 Exerciţii (rezolvările la pagina 174)

1. Găsiţi o parantezare optimă a produsului unui şir de matrice al cărui şir de dimensiuni este $(5, 10, 3, 12, 5, 50, 6)$ (adică $5 \times 10, 10 \times 3$, etc).

2. Arătaţi că o parantezare completă a unei expresii cu n elemente are exact $n - 1$ perechi de paranteze.

3. Fie $R(i, j)$ numărul de accesări ale elementului $m(i, j)$ din tabloul m, de către algoritmul *InmultireMatrici*, pentru calcularea elementelor tabloului. Arătaţi că: $\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3-n}{3}$. (Indicaţie: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$).

4. Se dă o matrice subdiagonală. Se consideră toate sumele în care se adună elementele de pe fiecare linie aflate *sub* elementul adunat anterior sau *sub-şi-la-dreapta*. Se cere un algoritm care găseşte suma maximă.

   Exemplu:
   $$\begin{array}{cccc} \mathbf{5} & & & \\ \mathbf{2} & 3 & & \\ \mathbf{8} & 1 & 2 & \\ 5 & \mathbf{9} & 3 & 4 \end{array}$$

   Suma maximă este $S = 5 + 2 + 8 + 9$.

5. Folosind rezultatele lui *InmultireMatrici* să se descrie un algoritm care realizează înmulţirea matricilor în ordine optimă.

Capitolul 12

Teste de autoevaluare

1. Algoritmi de sortare prin metoda Divide et impera.
2. Folosind un subalgoritm pentru înmulţirea matricilor să se scrie algoritmul care ridică la puterea k o matrice pătratică de ordinul n.
3. Dându-se parcurgerile în preordine A, B, C, D, E, F, G şi în inordine C, B, E, D, A, G, F să se deseneze arborele binar corespunzător.

Soluţii

1. (Din oficiu: 1p)
   Metoda presupune parcurgerea a trei paşi:
   a) împărţirea datelor în mai multe părţi;
   b) rezolvarea (de obicei recursivă) a problemei pe fiecare "bucată" de date;
   c) combinarea rezultatelor parţiale în rezultatul final.

   (1p)

   Sortarea prin interclasare presupune:
   a) împărţirea şirului iniţial în două părţi;
   b) sortarea fiecărei jumătăţi;
   c) interclasarea bucăţilor sortate.

   MergeSort(a, p, q)
   dacă $p \geq q$ *atunci*
   Return
   sfârşit dacă

$r = \left[\frac{p+q}{2}\right]$
MERGESORT(a, p, r)
MERGESORT$(a, r+1, q)$
Interclasare(a, p, r, q)
Return

(1p)

Interclasare(a, p, r, q)
$i = p$
$j = r + 1$
$k = 0$
repetă
  $k = k + 1$
  dacă $a(i) < a(j)$ *atunci*
    $b(k) = a(i)$
    $i = i + 1$
  altfel
    $b(k) = a(j)$
    $j = j + 1$
  sfârşit dacă
până când $i > r$ sau $j > q$
pentru $il = i, r$
  $k = k + 1$
  $b(k) = a(il)$
sfârşit pentru
pentru $jl = j, q$
  $k = k + 1$
  $b(k) = a(jl)$
sfârşit pentru
$k = 0$
pentru $i = p, q$
  $k = k + 1$
  $a(i) = b(k)$
sfârşit pentru
Return

(2p)

Apelarea se face:

Start Sortare1
  citeşte($n, (a(i), i = 1, n)$)
  MERGESORT$(a, 1, n)$

scrie($(a(i), i = 1, n)$)
Stop

Complexitatea rezultă din:

$$T(n) = 2T\left(\frac{n}{2}\right) + a \cdot n$$

unde a este o constantă. Din teorema centrală (pagina 74) onbţinem $T(n) = \Theta(n \log n)$. (1p)

Altă metodă este Quicksort (în practică, cel mai bun algoritm de sortare):

Quicksort(a, p, q)
dacă $p < q$ *atunci*
  Partitionare(a, p, q, r)
  Quicksort(a, p, r)
  Quicksort$(a, r + 1, q)$
sfârşit dacă
Return

(1p)

unde

Partitionare(a, p, q, r)
$el = a(p)$
$i = p - 1$
$j = q + 1$
repetă
  repetă
    $i = i + 1$
  până când $a(i) \geq el$
  repetă
    $j = j - 1$
  până când $a(j) \leq el$
  dacă $i < j$ *atunci*
    $a(i) \leftrightarrow a(j)$
  altfel
    $r = j$
  sfârşit dacă
până când $i \geq j$
Return

(2p)

Se apelează ca şi celălalt.

Complexitatea este $\Theta(n^2)$, dar în cazul (frecvent) al împărţirii proporţionale a şirului avem $\Theta(n \log n)$. Se pot face îmbunătăţiri prin alegerea aleatoare al lui *el* (elementul pivot). (1p)

2. Vom scrie doi subalgoritmi: unul pentru inmulţirea a două matrice pătratice şi altul pentru ridicarea la putere a unei matrici.

   Din oficiu (1p).

   InmultireMatrici(a, b, c, n)
   pentru $i = 1, n$
   pentru $j = 1, n$
   $c(i, j) = 0$
   pentru $k = 1, n$
   $c(i, j) = c(i, j) + a(i, k) \cdot b(k, j)$
   sfârşit pentru
   sfârşit pentru
   sfârşit pentru
   Return

   (3p)

   PutereMatrice(a, n, k, c)
   pentru $i = 1, n$
   pentru $j = 1, n$
   $b(i, j) = a(i, j)$
   sfârşit pentru
   sfârşit pentru
   pentru $p = 2, k$
   InmultireMatrici(a, n, b, c)
   pentru $i = 1, n$
   pentru $j = 1, n$
   $b(i, j) = c(i, j)$
   sfârşit pentru
   sfârşit pentru
   sfârşit pentru
   Return

   (4p)

   Subalgoritmul PutereMatrice se apelează în felul următor:

   Start RidicareLaPutere
   citeşte$(\ k, n, ((a(i, j), j = 1, n), i = 1, n)\ )$
   PutereMatrice(a, n, k, c)

scrie($((c(i,j), j = 1, n), i = 1, n)$)

Stop

(1p)

Complexitatea este $\Theta(n^3 k)$. (1p)

Observaţie: Prin algoritmul ridicării la putere prezentat în capitolul 5, problema 3, se poate scrie un algoritm de complexitate $\Theta(n^3 \log k)$. Utilizând un subalgoritm care să efectueze înmulţirea a două matrce în timp $\Theta(n^{\log_2 7})$ (algoritmul lui Strassen, vezi [1]), se ajunge la un algoritm de complexitate $\Theta\left(n^{\log_2 7} \log k\right)$ (sau chiar $\Theta(n^{2.376} \log k)$, conform aceleiaşi surse bibliografice).

3. Din oficiu (1p). Arborele este dat în figura 12.1. Raţionamentul prin care se construieşte arborele se poate face după o strategie de tip divide et impera: A fiind primul nod care apare în parcurgerea în preordine, rezultă că el este rădăcina arborelui. Ceea ce se află la stânga lui A în şirul dat în parcurgerea în inordine este subarborele stâng al rădăcinii (la noi: parcurgerea în inordine a subarborelui stâng al lui A este C, B, E, D, iar la parcurgerea în preordine aceleaşi 4 simboluri apar în ordinea B, C, D, E; analog, pentru partea dreaptă – inordine: G, F, preordine: F, G). La fiecare pas se detectează la fel ca mai sus rădăcina fiecărui subarbore, după care descendenţii săi stâng şi drept. Când se ajunge la o secvenţă formată dintr–un singur nod, nu mai avem ce să determinăm: arborele este complet construit.

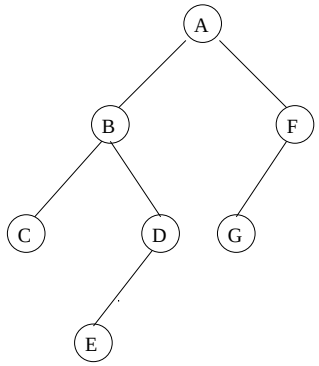


Figura 12.1: Arborele care corespunde celor două secvenţe: preordine A, B, C, D, E, F, G şi inordine C, B, E, D, A, G, F

Explicaţii: 4 puncte
Paşii efectuaţi până la sfârşit: 3p
Reprezentare arbore: 1p

Este posibil ca să se determine prin altă metodă un arbore care respectă doar una din secvenţele date. De exemplu, arborele din figura 12.2 produce doar parcurgerea în preordine conform secvenţei date, dar nu şi inordinea. (4p).

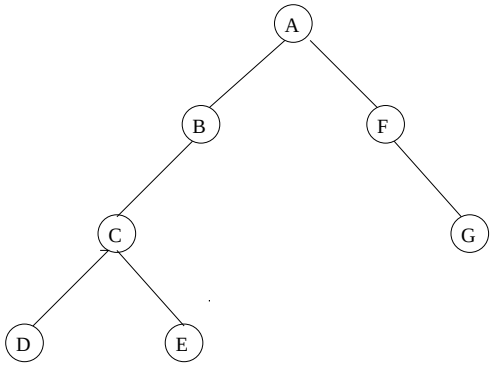


Figura 12.2: Arbore care produce la parcurgere în preordine A, B, C, D, E, F, G, dar nu dă rezultat corect pentru inordine: C, B, E, D, A, G, F, pe când la noi se obţine: D, C, E, B, A, F, G.

Anexa A

Metode utile în calculul complexităţii

A.1 Sume des întâlnite

1. Progresia aritmetică:

$$a_k = a_{k-1} + r,\ k = 2,\ a_1 \text{ şi } r \text{ daţi} \tag{A.1}$$

$$S = \sum_{k=1}^{n} a_k = \frac{(a_1 + a_n)n}{2} \tag{A.2}$$

2. Sume particulare:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2) \tag{A.3}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3) \tag{A.4}$$

$$\sum_{k=1}^{n} k^3 = \left[\frac{n(n+1)}{2}\right]^2 = \Theta(n^4) \tag{A.5}$$

şi în general:

$$\sum_{k=1}^{n} k^p = \Theta(n^{p+1}) \tag{A.6}$$

3. Progresii geometrice:

$$\begin{aligned} a_k &= a_{k-1} \cdot r,\ k = 1,\ a_1, r \neq 1 \text{ dați} \\ S &= \sum_{k=1}^{n} a_k = \sum_{k=1}^{n} a_1 r^k = a_1 r \sum_{k=0}^{n-1} r^k = a_1 r \cdot \frac{r^n - 1}{r - 1} \end{aligned} \quad (A.7)$$

Dacă $r < 1$, atunci

$$S = \sum_{k=1}^{n} a_k \leq \sum_{k=1}^{\infty} a_k = a_1 \cdot \frac{1}{1 - r} \quad (A.8)$$

A.2 Delimitarea sumelor

De multe ori, sumele nu se regăsesc în formele simple date mai sus. Există diferite tehnici de determinare a unei majorări şi minorări a acestora. Prezentăm câteva din ele mai jos.

A.2.1 Inducţia matematică

Una din modalităţile uşor de abordat pentru delimitarea unei sume este bazată pe inducţia matematică. De exemplu, să demonstrăm că seria aritmetică $\sum_{k=1}^{n} k$ are valoarea $\frac{1}{2}n(n+1)$. Putem verifica uşor pentru $n = 1$, $n = 2$ aşa că facem ipoteza de inducţie că formula este adevărată pentru orice n şi demonstrăm că are loc pentru $n + 1$. Avem

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = \frac{1}{2}n(n+1) + (n+1) = \frac{1}{2}(n+1)(n+2). \quad (A.9)$$

Nu întotdeauna putem determina valoarea exactă a unei sume. Pe noi oricum ne interesează ordinul sumei (deci constantele multiplicative şi termenii mici nu sunt întotdeauna esenţiali[1]). De exemplu, să demonstrăm că $\sum_{k=0}^{n} 2^k \leq c \cdot 2^n$ pentru o anumită constantă c, de la un rang n încolo (sau în altă notaţie: $\sum_{k=0}^{n} 2^k = O(2^n)$). Pentru condiţia iniţială $n = 0$ avem $\sum_{k=0}^{0} 2^k = 1 \leq c \cdot 1$, cât timp $c \geq 1$. Presupunând proprietatea de mărginire are loc pentru n să demonstrăm că are loc şi pentru $n + 1$. Avem:

$$\sum_{k=0}^{n+1} 2^k = \sum_{k=0}^{n} 2^k + 2^{n+1} \leq c \cdot 2^n + 2^{n+1} = \left(\frac{1}{2} + \frac{1}{c}\right) c \cdot 2^{n+1} \leq c \cdot 2^{n+1} \quad (A.10)$$

[1]Dar: $\Theta(2^{3n}) \neq \Theta(2^n)$; în acest caz constanta 3 nu poate fi neglijată

cât timp $(1/2 + 1/c) \leq 1$ sau, echivalent, $c \geq 2$. Astfel, $\sum_{k=0}^{n} 2^k = O(2^n)$, ceea ce am dorit să arătăm.

Atenţionăm cititorul asupra unui viciu de procedură în cazul utilizării acestui instrument matematic, care se regăşeste în următoarea demonstraţie "corectă": arătăm că $\sum_{k=1}^{n} k = O(n)$. Desigur, $\sum_{k=1}^{1} k = O(1)$. Acceptând delimitarea pentru n, o demonstrăm acum pentru $n+1$:

$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\ &= O(n) + (n+1) \quad \Leftarrow \textit{greşit !} \\ &= O(n+1).\end{aligned}$$

Greşeala în argumentaţie este aceea că O ascunde o "constantă" care creşte în funcţie de n şi deci nu este constantă. Ar trebui să arătăm că există o constantă care se ascunde în notaţia $O(\cdot)$ şi care este deci valabilă pentru toţi n.

A.2.2 Aproximarea prin integrale

Când o sumă poate fi exprimată ca $\sum_{k=m}^{n} f(k)$, unde $f(k)$ este o funcţie monoton crescătoare, o putem mărgini prin integrale:

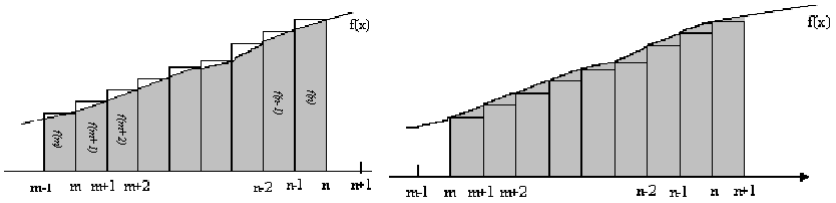
$$\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx. \tag{A.11}$$

Justificarea (geometrică, intuitivă) pentru această aproximare este arătată în figura A.1. Suma este reprezentată prin aria dreptunghiului din figură, iar integrala este regiunea haşurată de sub curbă. Când $f(k)$ este o funcţie monoton descrescătoare, putem utiliza o metodă similară pentru a obţine marginile

$$\int_{m}^{n+1} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx. \tag{A.12}$$

Aproximaţia integrală (A.12) dă o estimare strânsă pentru al n - lea număr armonic. Pentru o limită inferioară, obţinem

$$\sum_{k=1}^{n} \frac{1}{k} \geq \int_{1}^{n+1} \frac{dx}{x} = ln(n+1). \tag{A.13}$$



(a) Mărginire inferioară (b) Mărginire superioară

Figura A.1: Mărginire prin integrale.

Pentru limita superioară, deducem inegalitatea

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{dx}{x} = ln\ n, \tag{A.14}$$

care dă limita

$$\sum_{k=1}^{n} \frac{1}{k} \leq ln\ n + 1. \tag{A.15}$$

Avantajul acestei tehnici este că permite atât determinarea unei mărginiri inferioare, cât şi a unei mărginiri superioare, care pot duce la o exprimare a complexităţii cu notaţia Θ (mai exactă decât O sau Ω).

Propunem cititorului să aplice această tehnică pentru sumele prezentate anterior

A.3 Analiza algoritmilor recursivi

Analiza unui algoritm recursiv implică rezolvarea unui sistem de recurenţe. Vom vedea în continuare cum pot fi rezolvate astfel de recurenţe. Începem cu tehnica cea mai banală.

A.3.1 Metoda iteraţiei

Tehnica următoare cere ceva mai multă tehnică matematică, dar rezultatele sunt destul de expresive. Traiectoria ar fi: se execută câţiva paşi, se intuieşte forma generală, iar apoi se demonstrează prin inducţie matematică că această formă este corectă. Să considerăm, de exemplu, recurenţa problemei turnurilor din Hanoi. Pentru un $n > 1$ obţinem succesiv

$$t(n) = 2t(n-1) + \Theta(1) = 2^2 t(n-2) + \Theta(2) + \Theta(1) = \ldots = 2^{n-1} t(1) + \sum_{i=0}^{n-2} 2^i. \tag{A.16}$$

Rezultă $t(n) = 2^n - 1$. Prin inducţie matematică se demonstrează acum cu uşurinţă că această formulă este corectă.

A.3.2 Recurenţe liniare omogene

Vom prezenta modul de rezolvare al unor ecuaţii recurente liniare omogene, adică de forma:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \tag{A.17}$$

unde t_i sunt valorile pe care le căutăm, iar coeficienţii a_i sunt constante.

Vom ataşa recurenţei (A.17) ecuaţia:

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0 \tag{A.18}$$

Soluţiile acestei ecuaţii sunt fie soluţia trivială $x = 0$, care nu ne interesează, fie soluţiile ecuaţiei de grad k

$$a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0 \tag{A.19}$$

care este ecuaţia caracteristică a recurenţei (A.17).

Presupunând deocamdată că cele k rădăcini $r_1, r_2, \ldots, r_k$ ale acestei ecuaţii caracteristice sunt distincte, orice combinaţie liniară

$$t_n = \sum_{i=1}^{k} c_i r_i^n \tag{A.20}$$

este o soluţie a recurenţei (A.17), unde constantele $c_1, c_2, \ldots, c_k$ sunt determinate de condiţiile iniţiale.

Să exemplificăm prin recurenţa care defineşte şirul lui Fibonacci:

$$t_n = t_{n-1} + t_{n-2}\ , n \geq 2 \tag{A.21}$$

iar $t_0 = 0, t_1 = 1$. Putem să rescriem această recurenţă sub forma

$$t_n - t_{n-1} - t_{n-2} = 0 \tag{A.22}$$

care are ecuaţia caracteristică

$$x^2 - x - 1 = 0 \tag{A.23}$$

cu rădăcinile $r_{1,2} = (1 \pm \sqrt{5})/2$. Soluţia generală are forma

$$t_n = c_1 r_1^n + c_2 r_2^n. \tag{A.24}$$

Impunând condiţiile iniţiale, obţinem

$$\begin{aligned} c_1 + c_2 &= 0, \quad n = 0 \\ r_1 c_1 + r_2 c_2 &= 1, \quad n = 1 \end{aligned}$$

de unde determinăm

$$c_{1,2} = \pm 1/\sqrt{5}.$$

Deci, $t_n = 1/\sqrt{5}(r_1^n - r_2^n)$. Observăm că $r_1 = \phi = (1+\sqrt{5})/2,\ r_2 = -\phi^{-1}$ şi obţinem

$$t_n = 1/\sqrt{5}(\phi^n - (-\phi)^{-n})$$

Ca atare, implementarea recursivă pentru determinarea celui de-al n–lea termen al şirului lui Fibonacci duce la o complexitate exponenţială[2].

Dacă rădăcinile ecuaţiei caracteristice nu sunt distincte, se procedează după cum urmează: dacă r este o rădăcină de multiplicitate m a ecuaţiei caracteristice, atunci $t_n = r^n, t_n = nr^n, t_n = n^2 r^n, \ldots, t_n = n^{m-1} r^n$ sunt soluţii pentru (A.17). Soluţia generală pentru o astfel de recurenţă este atunci o combinaţie liniară a acestor termeni şi a termenilor proveniţi de la celelalte rădăcini ale ecuaţiei caracteristice. Ca mai sus, trebuie determinat exact k constante din condiţiile iniţiale.

Vom da din nou un exemplu. Fie recurenţa

$$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}, \quad n \geq 3 \tag{A.25}$$

iar $t_0 = 0, t_1 = 1, t_2 = 2$. Ecuaţia caracteristică are rădăcinile 1 (de multiplicitate 1) şi 2 (de multiplicitate 2). Soluţia generală este:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n.$$

Din condiţiile iniţiale, obţinem $c_1 = -2,\ c_2 = 2,\ c_3 = -1/2$.

A.3.3 Recurenţe liniare neomogene

Considerăm acum recurenţe de următoarea formă mai generală

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \tag{A.26}$$

unde b este o constantă, iar $p(n)$ este un polinom în n de grad d

Pentru a rezolva (A.26), este suficient să luăm următoarea ecuaţie caracteristică:

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x - b)^{d+1} = 0. \tag{A.27}$$

[2]Pentru o complexitatea logaritmică, a se vedea capitolul 5.

Odată ce s-a obţinut această ecuaţie, se procedează ca în cazul omogen.

Vom rezolva acum recurenţa corespunzătoare problemei turnurilor din Hanoi:

$$t_n = 2t_{n-1} + 1, \quad n \geq 1$$

iar $t_0 = 0$. rescriem recurenţa astfel

$$t_n - 2t_{n-1} = 1$$

care este de forma (A.26) cu $b = 1$ şi $p(n) = 1$, un polinom de gradul 0. Ecuaţia caracteristică este atunci $(x-2)(x-1) = 0$, cu soluţiile $1, 2$. Soluţia generală a recurenţei este

$$t_n = c_1 1^n + c_2 2^n.$$

Avem nevoie de două condiţii iniţiale. Ştim că $t_0 = 0$; pentru a găsi cea de-a doua condiţie calculăm

$$t_1 = 2t_0 + 1.$$

Din condiţiile iniţiale, obţinem

$$t_n = 2^n - 1.$$

din care se deduce că $t_n = \Theta(2^n)$.

Anexa B

Rezolvări

B.1 Rezolvări pentru capitolul 3

1. Problema 1.

   (a) Exemplul 1. acest program conţine doar instrucţiuni elementare care cer un timp constant de execuţie. Ca atare, timpul total este mărginit superior de o constantă, fapt figurat prin notaţia $\Theta(1)$.

   (b) Exemplul 2: justificarea este aceeaşi cu cea de la punctul precedent, deci complexitatea este $\Theta(1)$.

   (c) Exemplul 3: citirea datelor de intrare are un cost de $t_1(n) = c_1 \cdot (n+1)$, unde c_1 este o constantă reprezentând timpul necesar citirii unei valori. Atribuirea $s = 0$ se execută într-un timp constant $t_2 = c_2$, independent de dimensiunea datelor de intrare. Ciclarea înseamnă $t_3 = c_3 \cdot n$, unde c_3 este timpul necesar pentru calculul sumei si efectuarea atribuirii; de asemenea, c_3 nu depinde de n. Afişarea se face în timpul $t_4 = c_4$ (constant).
   Ca atare,

   $$T(n) = t_1(n) + t_2(n) + t_3(n) + t_4(n) = c_1 \cdot (n+1) + c_2 + c_3 \cdot n + c_4$$

   Conform proprietăţii 5, pagina 25, avem că:

   $$\begin{aligned} T(n) &= \Theta\left(c_1 \cdot (n+1) + c_2 + c_3 \cdot n + c_4\right) \\ &= \Theta(\max(c_1 \cdot (n+1) + c_3 \cdot n), c_2 + c_4) \\ &= \Theta(c_1 \cdot (n+1) + c_2 \cdot n) = \Theta((c_1 + c_3)n + c_1) \\ &= \Theta((c_1 + c_2)n) = \Theta(n), \end{aligned}$$

   ultima egalitate având loc datorită proprietăţii 7, pagina 25.

(d) Exemplul 4: citirea datelor este $t_1(n) = c_1(n+1)$; atribuirile înseamnă $t_2(n) = 3 \cdot c_2$, unde c_2 este timpul necesar unei atribuiri. Ciclul cu numărator înseamnă $t_2(n) = c_3 \cdot n$, timpul c_3 fiind constant din cauză că se fac evaluări de expresii logice şi atribuiri, costul fiecărei operaţii în parte fiind independent de n.
Deci

$$T(n) = t_1(n)+t_2(n)+t_3(n) = \Theta\left(c_1(n+1) + 3 \cdot c_2 + c_3 \cdot n\right) = \Theta(n)$$

O altă justificare: şirul este parcurs o singură dată, pentru fiecare element al său efectuându–se operaţii cu cost $\Theta(1)$; în total complexitatea este $\Theta(n)$.

(e) Exemplul 5: analiza algoritmului lui Euclid este destul de dificilă. Complexitatea algoritmului este $\Theta\left(\log\left(\min(a,b)\right)\right)$. A se vedea de exemplu [2].

(f) Exemplul 6: corpul ciclului cel mai interior se execută în timp constant deoarece conţine doar instrucţiuni elementare (atribuire, afişare). Se execută de un număr constant de ori, deci ciclul *repetă* ... *până când* $k = 5$ se execută într-un timp constant. La fel este şi cu ciclul *repetă* ... *până când* $j = 4$: corpul său se excută în timp constant, de un număr constant de ori. Idem în cazul ciclului cu contor exterior *pentru*. Ca atare, timpul total de execuţie este mărginit superior de o constantă, complexitatea fiind $\Theta(1)$. Propunem cititorului scrierea unui algoritm pentru listarea combinărilor câte 3 din mulţimea $\{1, 2, \ldots, n\}$. Care este în acest caz complexitatea?

2. Problema 2:

Start ProdusScalar
citeşte(n, $(a(i), i = 1, n)$, $(b(i), i = 1, n)$)
$ps = 0$ {0 este element neutru pentru adunare}
pentru $i = 1, n$
{dacă pasul este 1, se poate omite}
$ps = ps + a(i) \cdot b(i)$
sfârşit pentru
scrie('Produsul scalar este: ', ps)
Stop

Complexitatea este $\Theta(n)$.

3. Problema 3.
Ideea este de a parcurge şirul şi a compara fiecare element cu x; dacă

acest element este găsit, atunci este oprită căutarea, altfel se merge la următorul element. La sfârşit se raportează rezultatul.
Pentru a semnala faptul că x a fost găsit, vom folosi o variabilă booleană *gasit* care va primi valoarea *adevarat* dacă x a fost găsit în şir, $fals$ în caz contrar.

Start CăutareLiniară
  citeşte($n, (a(i), i = 1, n), x$)
  $gasit = fals$ {deocamdată nu am găsit pe x în a}
  $i = 1$ {i este indicele elementului curent din şir}
  repetă
    dacă $a(i) = x$ *atunci*
      $gasit = adevarat$
      $pozitie = i$ {*pozitie* este pozitia elementului x în şir}
    altfel
      $i = i + 1$
    sfârşit dacă
  până când $i > n$ sau $(gasit = adevarat)$
  dacă $gasit = adevarat$ *atunci*
    scrie('Am găsit ', x, ' pe poziţia ', *pozitie*)
  altfel
    scrie(x, ' nu a fost găsit ')
  sfârşit dacă
Stop

Complexitatea este $\Theta(n)$, cazul cel mai defavorabil fiind când x nu se află în şir.

Observaţie: Deoarece execuţia instrucţiunilor din interiorul ciclului trebuie să se facă cel puţin o dată, este corectă utilizarea unui ciclu *repetă*.

4. Problema 4: vom folosi un algoritm asemănător cu cel de la problema precedentă, folosind informaţia că şirul este ordonat crescător. Vom da două variante, cu analiză de complexitate.

   (a) Prima variantă: începem căutarea, dar vom impune o condiţie suplimentară: trebuie ca permanent $x \leq a(i)$. Dacă avem $x > a(i)$, atunci este clar că x nu mai poate fi găsit la dreapta indicelui i.

   Start CăutareLiniarăCrescător
     citeşte($n, (a(i), i = 1, n), x$)
     $gasit = fals$ {deocamdată nu am găsit pe x în a}
     $i = 1$ {i este indicele elementului curent din şir}

repetă
dacă $a(i) = x$ *atunci*
$gasit = adevarat$
$pozitie = i$ {*pozitie* este pozitia elementului x în şir}
altfel
$i = i + 1$
sfârşit dacă
până când $i > n$ sau $gasit = adevarat$ sau $x > a(i)$
dacă $gasit = adevarat$ *atunci*
scrie('Am găsit ', x, ' pe poziţia ', *pozitie*)
altfel
scrie(x, ' nu a fost găsit ')
sfârşit dacă
Stop

Complexitatea este aceeaşi ca la algoritmul precedent: $\Theta(n)$.
Observaţie Este esenţial ca testarea $i > n$ să se facă înainte de testarea $a(i) \leq x$; dacă spre exemplu locul lor ar fi fost inversat, atunci era posibil ca să avem $i = n + 1$, iar condiţia $a(i) \leq x$ să fie testată, deşi $a(n + 1)$ nu există.

(b) A doua variantă speculează mai inteligent faptul că şirul este gata sortat. Putem împărţi şirul în două subşiruri de dimensiuni cât mai apropiate. Se testează dacă elementul de la mijloc este egal cu x. Dacă da, atunci căutarea se opreşte cu succes. Dacă nu, şi $x <$elementul din mijloc, atunci căutarea va continua să se facă în prima jumătate a şirului (e clar că în a doua jumătate nu poate fi găsit); dacă x este mai mare decât elementul de la jumatate, atunci căutarea se va continua în a doua jumătate. Căutarea continuă până când fie se găseşte x în şir, fie nu mai avem nici un element de cercetat, caz care înseamnă căutare fără succes.
Acest algoritm se numeşte *algoritmul căutării binare (prin înjumătăţire).*

Start CăutareBinară
citeşte($n, (a(i), i = 1, n), x$)
$gasit = fals$ {variabila *gasit* are aceeaşi semnificaţie ca în algoritmul anterior}
$stanga = 1$
$dreapta = n$
{*stanga* şi *dreapta* reprezintă capetele între care se face căutarea lui x}
{Initial, căutarea se face în tot vectorul}

cât timp $gasit = fals$ şi $stanga \leq dreapta$
{nu l-am găsit pe x şi mai avem unde căuta}
$mijloc = \left[\frac{stanga+dreapta}{2}\right]$ {mă pozitionez la jumătatea şirului}
dacă $a(mijloc) = x$ *atunci*
$gasit = adevarat$
altfel
dacă $x < a(mijloc)$ *atunci*
$dreapta = mijloc - 1$
altfel
$stanga = mijloc + 1$
sfârşit dacă
{caut doar în jumătatea corespunzătoare}
sfârşit dacă
sfârşit cât timp
dacă $gasit = adevarat$ *atunci*
scrie('Am găsit ', x, ' pe poziţia ', *pozitie*)
altfel
scrie(x, ' nu a fost găsit ')
sfârşit dacă
Stop

Complexitate: Timpul $T(n)$ pentru rezolvarea problemei de dimensiune n satisface ecuaţia: $T(n) = T(n/2)+c$, c fiind o constată necesară testărilor şi atribuirilor (operaţii elementare, al căror cost în parte nu depinde de n). Conform teoremei centrale (enunţată la capitolul "Divide et impera") obţinem că

$$T(n) = \Theta\left(\log_2(n)\right),$$

adică mult mai performant decât în varianta 4a.

5. Problema 4: presupunem că minimul este primul element din şir. Luăm apoi fiecare din celelalte elemente ale vectorului a şi le comparăm cu minimul curent cunoscut: dacă acest element este mai mic decât minimul, atunci e clar că am descoperit un minim "mai bun", deci noul minim va fi elementul curent. Procedeul continuă până când se epuizează elementele din şir. La sfârşit minimul găsit va fi de fapt minimul absolut al şirului, ceea ce trebuia determinat.

Start Minim
citeşte($n, (a(i), i = 1, n)$)
$minim = a(1)$
pentru i=2,n

dacă $a(i) < minim$ *atunci*
$minim = a(i)$
sfârşit dacă
sfârşit pentru
scrie('minimul este ', $minim$)
Stop

Numărul de comparaţii este $n - 1$. Nu se poate coborî sub aceasta (demonstraţia se face prin inducţie).
Observaţie. Există varianta de a iniţializa $minim$ cu o valoare foarte mare, considerată $+\infty$ pentru tipul de date care este folosit. De multe ori limbajele de programare pun la dispoziţie constante reprezentând valorile cele mai mari (mici) repezentabile pe un anumit tip de date numeric (Pascal, C/C++, Java, C#).

6. Problema 5: vom folosi un contor (o variabilă) pentru a urmări de câte ori apare minimul.

   (a) Vom rezolva problema în doi paşi: vom determina minimul printr-o parcurgere a şirului, ca mai sus, iar apoi vom parcurge şirul încă o dată pentru a număra de câte ori a apărut acesta.

   Start ContorMinim1
   citeşte($n, (a(i), i = 1, n)$)
   $minim = a(1)$
   pentru i=2,n
   dacă $a(i) < minim$ *atunci*
   $minim = a(i)$
   sfârşit dacă
   sfârşit pentru
   $contor = 0$
   pentru $i = 2, n$
   dacă $minim = a(i)$ *atunci*
   $contor = contor + 1$
   sfârşit dacă
   sfârşit pentru
   scrie('minimul este ', $minim$)
   scrie('apare de ', contor, ' ori')
   Stop

   Complexitate: $\Theta(n)$, deoarece şirul se parcurge de două ori, pentru fiecare element efectuându–se operaţii elementare ce nu depind de n (dimensiunea intrării).

(b) Vom rezolva problema printr–o singură parcurgere a şirului; complexitatea teoretică rămâne aceeaşi.

Start ContorMinim2
  citeşte($n, (a(i), i = 1, n)$)
  $minim = a(1)$
  $contor = 1$
  pentru $i = 2, n$
    dacă $minim > a(i)$ *atunci*
      $minim = a(i)$ {am găsit un minim "mai bun"}
      $contor = 1$ {resetăm contorul la 1}
    altfel
      dacă $minim = a(i)$ *atunci*
        $contor = contor + 1$ {am mai găsit o valoare egală cu minimul curent}
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
  scrie('minimul este ', $minim$)
  scrie('apare de ', $contor$, ' ori')
Stop

7. Problema 6: vom construi doi vectori: unul reţine numerele distincte din a, iar celălalt contorizează numărul de apariţii pentru fiecare număr. Vom scrie un subalgoritm de tip funcţie de căutare liniară a unui element x într-un vector v care conţine un anumit număr de elemente. Această funcţie returnează poziţia pe care se găseşte elementul x în v sau 0 în cazul în care x nu este cuprins în v:

CautaLiniar(x, v, l)
$pozitie = 0$
$gasit = fals$
$i = 1$
repetă
  dacă $x = v(i)$ *atunci*
    $gasit = adevarat$
    $pozitie = i$
  altfel
    $i = i + 1$
  sfârşit dacă
până când $i > l$ sau $gasit = adevarat$
$CautaLiniar = pozitie$
Return

Start CelMaiDes
citeşte($n, (a(i), i = 1, n)$)
$k = 0$ {k este numărul de elemente distincte din a}
pentru $i = 1, n$
$pozitieInDiferite$ =CautăLiniar $(a(i), diferite, k)$
dacă $pozitieInDiferite = 0$ *atunci*
$k = k + 1$ {$a(i)$ nu a mai fost găsit înainte}
{punem $a(i)$ în vectorul $diferite$}
$diferite(k) = a(i)$
$contor(k) = 1$
altfel
{$a(i)$ a mai apărut o înainte în a}
$contor(pozitieInDiferite) = contor(pozitieInDiferite)+1${l-
am mai găsit o dată}
sfârşit dacă
sfârşit pentru
{acum căutăm în vectorul *contor* să vedem care număr a apărut cel mai des}
$maximAparitii = contor(1)$
$pozMax = 1$
pentru $i = 2, k$
dacă $contor(i) > maximAparitii$ *atunci*
$maximAparitii = contor(i)$
$pozMax = i$
sfârşit dacă
sfârşit pentru
scrie('Numărul cel mai frecvent: ', $diferite(pozMax)$)
scrie('Apare de ', $maximAparitii$, ' ori')
Stop

Complexitate: cazul cel mai defavorabil este atunci când avem doar elemente distincte în şirul a. În acest caz, un apel *CautaLiniar*$(a(i), diferite, k)$ are costul de forma $a \cdot k + b$, a şi b constante. Ciclul din programul principal duce la un timp

$$T(n) = \sum_{k=1}^{n} (a \cdot k + b) = \Theta(n^2)$$

Exemplu: pentru $n = 4$, $a = (10, 20, 30, 40)$ vom avea:

$diferit =$	$(10, 20, 30, 40)$
$contor =$	$(1, 1, 1, 1)$
$k =$	4
cel mai frecvent:	10
frecvenţa:	1

8. Problema 8: vom profita de faptul că vectorul dat este sortat crescător. Vom căuta o secvenţă de numere constante, de lungime maximă.

   Start CelMaiDes
   $inceputSecventa = 1$
   $sfarsitSecventa = 1$
   $lungimeSecventaMaxima = 0$
   $suntInSir = adevarat$
   cât timp $suntInSir = adevarat$
   cât timp $sfarsitSecventa \leq n$ şi $a(inceputSecventa) = a(sfarsitSecventa)$

   $sfarsitSecventa = sfarsitSecventa + 1$
   sfârşit cât timp
   dacă $sfarsitSecventa - inceputSecventa > lungimeSecventaMaxima$
   atunci
   $apareMaxim = a(inceputSecventa)$
   $lungimeSecventaMaxima = sfarsitSecventa - inceputSecventa$
   sfârşit dacă
   $inceputSecventa = sfarsitSecventa$
   dacă $inceputSecventa > n$ *atunci*
   $suntInSir = fals$ {am parcusr tot şirul}
   sfârşit dacă
   sfârşit cât timp
   scrie('Numărul cel mai frecvent: ', $apareMaxim$)
   scrie('Apare de ', $lungimeSecventaMaxima$, ' ori')
   Stop

   Complexitate: se observă că valoarea $sfarsitSecventa$ ia pe rând valorile $1, 2, \ldots, n + 1$; de fiecare dată sunt efectuate operaţii elementare, deci avem $T(n) = \Theta(n + 1) = \Theta(n)$.
   Observaţie: Pentru a rezolva problema de la punctul anterior, am putea face o sortare a şirului, după care să aplicăm algoritmul de mai sus. Complexitatea totală ar fi: sortarea implică un timp $\Theta(n \log n)$, algoritmul anterior $\Theta(n)$, deci per total $\Theta(n \log n)$, sensibil mai bine decât $\Theta(n^2)$ care fusese obţinut anterior.

9. Problema 9:

Start Matrice
citeşte(n)
citeşte($((a(i,j), j = 1, n), i = 1, n)$)
{subpunctul a}
$p = 1$ {1 este element neutru pentru înmulţire}
pentru $i = 1, n$
pentru $j = 1, n$
dacă $a(i,j) \neq 0$ *atunci*
$p = p \cdot a(i,j)$
sfârşit dacă
sfârşit pentru
sfârşit pentru
scrie('Produsul elementelor nenule din matrice este: ', p)
{subpunctul b}
$p = 1$
pentru $i = 1, n$
dacă $a(i,i) \neq 0$ *atunci*
$p = p \cdot a(i,j)$
sfârşit dacă
sfârşit pentru
scrie('Produsul elementelor nenule de pe diagonala principală este: ', p)
{subpunctul c}
$p = 1$
pentru $i = 2, n$
pentru $j = 1, i-1$
dacă $a(i,j) \neq 0$ *atunci*
$p = p \cdot a(i,j)$
sfârşit dacă
sfârşit pentru
sfârşit pentru
scrie('Prod. elem. de sub diagonala principală este:', p)
Stop

Complexitate:

(a) $\Theta(n^2)$ (fiecare element al matricii este luat în considerare o singură dată, efectuându-se cu el operaţii elementare);

(b) $\Theta(n)$;

(c) $T(n) = \sum_{i=2}^{n}(i-1)c = c\sum_{i=2}^{n}(i-1) = c\frac{n(n-1)}{2}$, deci $T(n) = \Theta(n^2)$.

B.2 Rezolvări pentru capitolul 4

1. Problema 1: Vom folosi pentru memorarea elementelor din coadă un vector v de n elemente. Vom presupune că indicii vectorului sunt între 0 şi $n-1$. Deoarece coada este circulară, după locaţia de indice $n-1$ urmează locaţia de indice 0. Iniţial, coada este vidă, fapt semnalat prin $inceput = sfarsit = 0$.

   $\underline{x \Rightarrow Coada}$
   $sfarsit = (sfarsit + 1) mod\ n$
   $\underline{dacă}\ sfarsit = inceput\ \underline{atunci}$
   "Coada este plină"
   $\underline{altfel}$
   $v(sfarsit) = x$
   $\underline{sfârşit\ dacă}$
   $\underline{Return}$
   $\underline{x \Leftarrow Coada}$
   $\underline{dacă}\ inceput = sfarsit\ \underline{atunci}$
   "Coada este vidă"
   $\underline{altfel}$
   $inceput = (inceput + 1) mod\ n$
   $x = v(inceput)$
   $\underline{sfârşit\ dacă}$
   $\underline{Return}$

   Complexitatea fiecărui subalgoritm este $\Theta(1)$. Iniţial avem $inceput = sfarsit = 0$.
   Observaţie: Se folosesc efectiv doar $n-1$ locaţii ale vectorului; dar în acest mod se poate distinge între situaţiile de coadă plină şi coadă vidă.

2. Problema 2:

   (a) Parcurgerea în inordine, iterativ:

   InordineIterativ(rad)
   $i = rad$
   $S = \emptyset$\{ S este o stivă\}
   $\underline{repetă}$
   $\underline{cât\ timp}\ LS(i) \neq *$
   $i \Rightarrow S$
   $i = LS(i)$
   $\underline{sfârşit\ cât\ timp}$
   $\underline{scrie}(\ INFO(i)\ )$

cât timp $LD(i) = *$
  dacă $S = \emptyset$ *atunci*
    Return{ se iese din subalgoritm, deci şi din ciclul infinit}
  sfârşit dacă
  $i \Leftarrow S$
  scrie($INFO(i)$)
sfârşit cât timp
$i = LD(i)$
până când fals{ciclu infinit, din care se iese datorită lui Return}

(b) Parcurgerea în postordine, iterativ:

PostordineIterativ(rad)
$i = rad$
$S = \emptyset$\{ S este o stivă\}
repetă
  cât timp $LS(i) \neq' *$‘
    $i \Rightarrow S$
    $i = LS(i)$
  sfârşit cât timp
  cât timp $LD(i) =$ ‘$*'$
    repetă
      scrie($INFO(i)$)
      dacă $S = \emptyset$ *atunci*
        Return{se iese din subalgoritm, deci şi din ciclul infinit}
      sfârşit dacă
      $j = i$
      $i \Leftarrow S$
    până când $j = LS(i)$
  sfârşit cât timp
  $i \Rightarrow S$
  $i = LD(i)$
până când fals{ciclu infinit, din care se iese datorită lui Return}

Apelul subalgoritmilor de mai sus se face având drept parametru de apel rad, reprezentând adresa rădăcinii arborelui.
Complexitatea fiecăreia din proceduri este de $\Theta(n)$, unde n este numărul de vârfuri ale arborelui.

3. Problema 3:

Vom extrage succesiv vârful stivei S şi îl vom depune într-o altă stivă

T. Extragerile încetează fie când S devine vidă, fie când x este găsit. Elementele din T vor fi depuse înapoi în S.

Extrage(S, x)
$T = \emptyset$
$gasitx = fals$
cât timp $S \neq \emptyset$ şi $gasitx = fals$
  $a \Leftarrow S$ { scoatem vârful stivei}
  dacă $a \neq x$ *atunci*
    $a \Rightarrow T$ { îl depunem în T}
  altfel
    $gasitx = adevarat$
  sfârşit dacă
sfârşit cât timp
cât timp $T \neq \emptyset$
  $a \Leftarrow T$
  $a \Rightarrow S$
sfârşit cât timp
Return

Complexitate: cazul cel mai defavorabil este acela în care x nu se găseşte în S. În acest caz, tot conţinutul lui S este trecut în T şi apoi repus în S. Complexitatea este deci $\Theta(|S|)$, unde $|S|$ reprezintă numărul de elemente aflate iniţial în S.

Observaţie: Primul ciclu *cât timp* poate fi scris mai adecvat ca un ciclu *repetă*, având în vedere faptul că execuţia ciclui se face cel puţin o dată.

4. Problema 4:

Rezolvarea este asemănatoare cu cea de la punctul precedent.

Insereaza(S, x, y)
$T = \emptyset$
gasitx=fals
cât timp $S \neq \emptyset$ şi $gasitx = fals$
  $a \Leftarrow S$ { scoatem vârful stivei}
  dacă $a \neq x$ *atunci*
    $a \Rightarrow T$ { îl depunem în T}
  altfel
    $gasitx = adevarat$
    $a \Rightarrow S$
  sfârşit dacă
sfârşit cât timp
dacă $gasitx = adevarat$ *atunci*

$y \Rightarrow S$
sfârşit dacă
cât timp $T \neq \emptyset$
$a \Leftarrow T$
$a \Rightarrow S$
sfârşit cât timp
Return

Complexitatea este $\Theta(|S| + 1) = \Theta(|S|)$, motivaţia fiind asemănătoare cu cea de la punctul precedent.

5. Problema 5:

Vom folosi două stive: $Stiva1, Stiva2$.

$x \Rightarrow Coada$
dacă $Stiva1 = \emptyset$ *atunci*
Goleste($Stiva2, Stiva1$){Goleşte elementele din $Stiva2$ în $Stiva1$}
sfârşit dacă
$x \Rightarrow Stiva1$
Return
$x \Leftarrow Coada$
dacă $Stiva2 = \emptyset$ *atunci*
Goleste($Stiva1, Stiva2$){Goleşte elementele din $Stiva1$ în $Stiva2$}
sfârşit dacă
dacă $Stiva2 = \emptyset$ *atunci*
"Coada este vida"
altfel
$x \Leftarrow Stiva2$
sfârşit dacă
Return

Subalgoritmul Goleste(A, B) unde A şi B sunt stive va trece toate elementele din A în B, prin extrageri repetate.

Goleste(A, B)
cât timp $A \neq \emptyset$
$x \Leftarrow A$
$x \Rightarrow B$
sfârşit cât timp
Return

Numărul de operaţii de inserare şi extragere în fiecare caz zunt:

(a) Subalgoritmul Goleste(A, B) efectuează $2|A|$ extrageri ($|A|$ este

numărul de elemente din stiva A);

(b) Subalgoritmul $x \Rightarrow Coada$ efectuează $2|Stiva2|+1 = 2|Coada|+1$ operaţii de inserare/extragere;

(c) Subalgoritmul $x \Leftarrow Coada$ efectuează $2|Stiva1|+1 = 2|Coada|+1$ operaţii de inserare/extragere;

6. Problema 6:

Vom folosi două cozi: $Coada1$, $Coada2$.

$x \Rightarrow Stiva$
$x \Rightarrow Coada1$
Return
$x \Leftarrow Stiva$
dacă $Coada1 = \emptyset$ *atunci*
  "eroare: stiva vidă"
altfel
  cât timp $Coada1 \neq \emptyset$
    $x \Leftarrow Coada1$
    dacă $Coada1 \neq \emptyset$ *atunci*
      {x nu este ultimul element din $Coada1$}
      $x \Rightarrow Coada2$
    sfârşit dacă
  sfârşit cât timp{Goleşte $Coada2$ în $Coada1$}
  cât timp $Coada2 \neq \emptyset$
    $y \Leftarrow Coada2$
    $y \Rightarrow Coada1$
  sfârşit cât timp
sfârşit dacă
Return

Numărul de operaţii de inserare/extragere pentru fiecare din cei doi subalgoritmi sunt:

(a) Subalgoritmul $x \Rightarrow Stiva$ efectuează o inserare

(b) Subalgoritmul $x \Leftarrow Stiva$ are numărul de operaţii cerut egal cu $2(|Coada1| - 1) + 1$ (primul ciclu) adunat cu $2(|Coada1| - 1)$, în total $4|Coada1| - 3 = 4|Stiva| - 3$.

Sugerăm cititorului să găsească şi alte solucţii pentru această problemă.

B.3 Rezolvări pentru capitolul 5

1. Problema 1. Definim maximul unui şir $x = (x(1), x(2), \ldots, x(n))$ sub formă recursivă astfel:

$$maxim(x(1 \ldots n)) = \begin{cases} x(1), & \text{dacă n=1} \\ \max\{x(n), x(1 \ldots n-1)\}, & \text{dacă } n > 1 \end{cases}$$

unde $\max\{a, b\}$ este maximul dintre a şi b. Algoritmul este dat mai jos:

Start MaximRec
  citeşte($n, (x(i), i = 1, n)$)
  $M = Maxim(x, n)$
  scrie(M)
Stop

unde subalgoritmul recursiv $Maxim$ este:

$Maxim(x, n)$
dacă $n = 1$ *atunci*
  $MaximRec = x(1)$
altfel
  $MaximRec = \max\{x(n), Maxim(x, n-1)\}$
sfârşit dacă
Return

Complexitatea este dată de ecuaţia $T(n) = T(n-1)+c$ care are soluţia $T(n) = \Theta(n)$.

2. Problema 2. Definim suma elementelor unu şir $x = (x(1), \ldots, x(n))$ sub formă recursivă astfel:

$$suma(x(1 \ldots n)) = \begin{cases} 0, & \text{dacă n=0} \\ x(n) + suma(x(1 \ldots n-1)), & \text{dacă } n > 0 \end{cases}$$

Start SumaRec
  citeşte($n, (x(i), i = 1, n)$)
  $S = Suma(x, n)$
  scrie(S)
Stop

unde subalgoritmul recursiv $Suma$ este:

$Suma(x, n)$
dacă $n = 0$ *atunci*

$Suma = 0$
altfel
$Suma = x(n) + Suma(x, n-1)$
sfârşit dacă
Return

Complexitatea este dată de ecuaţia $T(n) = T(n-1)+c$ care are soluţia $T(n) = \Theta(n)$.

3. Problema 3. Vom da o formulă recursivă de calcul a puterii naturale a unui număr.

$$a^n = \begin{cases} 1, & \text{dacă } n = 0 \\ a, & \text{dacă } n = 1 \\ \left(a^{n/2}\right)^2, & \text{dacă } n > 1, n \text{ par} \\ a \cdot \left(a^{\lfloor n/2 \rfloor}\right)^2, & \text{dacă } n > 1, n \text{ impar} \end{cases}$$

Subalgoritmul recursiv este:

Putere(a, n)
dacă $n = 0$ *atunci*
$Putere = 1$
Return
sfârşit dacă
dacă $n = 1$ *atunci*
$Putere = a$
Return
sfârşit dacă
$temp = Putere(a, \lfloor n/2 \rfloor)$
dacă $n \mod 2 = 0$ *atunci*
{n este par}
$Putere = temp$
altfel
$Putere = a \cdot temp$
sfârşit dacă
Return

Complexitatea este dată de ecuaţia $T(n) = T(n/2)+c$, de unde $T(n) = \Theta(\log n)$ (rezultat al teoremei 1, pagina 74).

4. Problema 4. Pentru a^{15}, conform algoritmului precedent, avem:

$$a^{15} = \left(\left((a)^2 \cdot a\right)^2 \cdot a\right)^2 \cdot a$$

deci în total 6 înmulţiri.

Dar a^{15} se poate calcula astfel: se calculează termenii $a^2, a^3 = a^2 \cdot a, a^6 = a^3 \cdot a^3, a^{12} = a^6 \cdot a^6, a^{15} = a^{12} \cdot a^3$, deci cu 5 înmulţiri.

5. Problema 5. Să considerăm matricea:

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

Calculând produsul:

$$A \cdot \begin{pmatrix} f_{n-1} \\ f_n \end{pmatrix}$$

obţinem matricea

$$\begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$$

de unde deducem

$$\begin{pmatrix} f_{n-1} \\ f_n \end{pmatrix} = A^{n-1} \cdot \begin{pmatrix} f_0 \\ f_1 \end{pmatrix}$$

Modificând algoritmul de la problema 3 să ridice la putere o matrice în loc de un scalar, obţinem termenul f_n în timp $\Theta(\log n)$.

6. Problema 6. Algoritmuii recusrivi pentru parcurgeri sunt următorii:

   (a) Parcurgerea în preordine se efectuează astfel: se prelucrează informaţia rădăcinii, apoi se prelucrează recursiv subarborele stâng, apoi subarborele drept.

   Preordine(Rad)
   dacă $Rad \neq *$ *atunci*
   scrie($Info(Rad)$){Sau orice alt subalgoritm de prelucrarea a informaţiei din Rad}
   Preordine($LS(Rad)$)
   Preordine($LD(Rad)$)
   sfârşit dacă
   Return

   Apelul acestui subalgoritm se face cu *Preordine(Rad)*, unde *Rad* reprezintă rădăcina subarborelui.

   (b) Parcurgerea recursivă în postordine se efectuează astfel: se parcurge recursiv subarborele stâng, apoi subarborele drept, apoi se prelucrează informaţia din rădăcină.

Postordine(Rad)
dacă $Rad \neq *$ *atunci*
  Postordine($LS(Rad)$)
  Postordine($LD(Rad)$)
  scrie($Info(Rad)$){Sau orice alt subalgoritm de prelucrarea a informaţiei din Rad}
sfârşit dacă
Return

Apelul se face ce în cazul anterior.

(c) Parcurgerea recursivă în inordine se efectuează astfel: se parcurge recursiv subarborele stâng, apoi se prelucrează informaţia din rădăcină, apoi se parcurge recursiv subarborele drept.

Inordine(Rad)
dacă $Rad \neq *$ *atunci*
  Inordine($LS(Rad)$)
  scrie($Info(Rad)$){Sau orice alt subalgoritm de prelucrarea a informaţiei din Rad}
  Inordine($LD(Rad)$)
sfârşit dacă
Return

Apelul se face ce în cazul anterior.

B.4 Rezolvări pentru capitolul 6

1. Problema 1.

   Vom da două variante de rezolvare: iterativă şi recursivă. Ambele variante vor folosi doi vectori de sărituri: $sarituraI$ şi $sarituraJ$ cu câte 8 elemente, reprezentând cele 8 deplasări de la poziţia curentă: de exemplu, dacă poziţia curentă este pe linia i şi coloana j, atunci prima săritură va duce calul la coordonatele $(i+sarituraI(1), j+sarituraJ(1)$ (cu condiţia să nu se iasă din perimetrul tablei).

   (a) Varianta iterativă: pentru refacerea pasului "înapoi" vom folosi o matrice pătratică $deLa$ de ordinul n, care va reţine pentru fiecare celulă de coordonate (i, j) indicele săriturii care a determinat atingerea poziţiei respective (vom putea şti astfel de unde s–a sărit în poziţia curentă). Soluţia se dă sub forma unei matrice pătratice de ordinul n conţinând în fiecare celulă de coordonate (i, j) indicele săriturii ce se face în continuare (cu excepţia ultimei celule în care se sare).

```
Start CalSah
  citeşte( n )
  sariturаI(1) = +1, sariturаJ(1) = +2
  sariturаI(2) = −1, sariturаJ(2) = +2
  sariturаI(3) = +1, sariturаJ(3) = −2
  sariturаI(4) = −1, sariturаJ(4) = −2
  sariturаI(5) = +2, sariturаJ(5) = +1
  sariturаI(6) = −2, sariturаJ(6) = +1
  sariturаI(7) = +2, sariturаJ(7) = −1
  sariturаI(8) = −2, sariturаJ(8) = −1
  pentru i = 1, n
    pentru j = 1, n
      deLa(i, j) = 0
      tabla(i, j) = 0
    sfârşit pentru
  sfârşit pentru
  nrSaritura = 1
  i = j = 1
  cât timp nrSaritura > 0
    dacă nrSaritura = n^2 atunci
      ScrieSolutie(tabla, n)
      indiceSariturаInapoi = deLa(i, j)
      tabla(i, j) = 0
      i = i − sariturаI(indiceSariturаInapoi)
      j = j − sariturаJ(indiceSariturаInapoi)
      nrSaritura = nrSaritura − 1
    sfârşit dacă
    dacă tabla(i, j) < 8 atunci
      tabla(i, j) = tabla(i, j) + 1
      iInainte = i + sariturаI(tabla(i, j))
      jInainte = j + sariturаJ(tabla(i, j))
      dacă PoateSari(iInainte, jInainte, tabla, n)=adevarat atunci
        deLa(iInainte, jInainte) = tabla(i, j)
        i = iInainte
        j = jInainte
        nrSaritura = nrSaritura + 1
      sfârşit dacă
    altfel
      dacă nrSaritura > 1 atunci
        iInapoi = i − sariturаI(deLa(i, j))
        jInapoi = j − sariturаJ(deLa(i, j))
```

```
            tabla(i, j) = 0
            i = iInapoi
            j = jInapoi
          sfârşit dacă
          nrSaritura = nrSaritura − 1
        sfârşit dacă
      sfârşit cât timp
    Stop
    ScrieSolutie(tabla, n)
    pentru i = 1, n
      pentru j = 1, n
        scrie( tabla(i, j) )
      sfârşit pentru
    sfârşit pentru
    Return
    PoateSari(linie, coloana, tabla, n)
    dacă linie > 0 şi linie < n + 1 şi coloana > 0 şi coloana < n + 1 şi
    tabla(linie, coloana) = 0 atunci
      rezultat = adevarat
    altfel
      rezultat = fals
    sfârşit dacă
    PoateSari = rezultat
    Return
```

(b) Varianta recursivă

```
    Cal(i, j, n, nrSaritura, tabla)
    dacă nrSaritura = n · n atunci
      ScrieSolutie(tabla, n)
    altfel
      pentru indiceSaritura = 1, 8
        iInainte = i + sariturai[indiceSaritura]
        jInainte = j + sariturai[indiceSaritura]
        dacă PoateSari(iUrmator, jUrmator, tabla, n)=adev atunci
          tabla[iInainte, jInainte] = nrSaritura + 1
          Cal(iInainte, jInainte, n, nrSaritura + 1, tabla)
          tabla[iInainte, jInainte] = 0
        sfârşit dacă
      sfârşit pentru
    sfârşit dacă
    Return
```

Subalgoritmii ScrieSolutie şi PoateSari sunt identici cu cei de la varianta iterativă.

Start CalSah
  citeşte(n)
  $sarituraI(1) = +1, sarituraJ(2) = +2$
  $sarituraI(2) = -1, sarituraJ(2) = +2$
  $sarituraI(3) = +1, sarituraJ(3) = -2$
  $sarituraI(4) = -1, sarituraJ(4) = -2$
  $sarituraI(5) = +2, sarituraJ(5) = +1$
  $sarituraI(6) = -2, sarituraJ(2) = +1$
  $sarituraI(7) = +2, sarituraJ(2) = -1$
  $sarituraI(1) = -2, sarituraJ(2) = -1$
  pentru $i = 1, n$
    pentru $j = 1, n$
      $tabla(i, j) = 0$
    sfârşit pentru
  sfârşit pentru
  $tabla(1, 1) = 1$
  $Cal(1, 1, n, 1, tabla)$
Stop

2. Problema 2.

S–a demonstrat cu ajutorul calculatorului că 4 culori sunt suficiente pentru a colora cu restricţiile date orice hartă. Vom da două variante, iterativă şi recursivă. În ambele variante vectorul *culoare* de n elemente va conţine culorile ţărilor.

(a) Varianta iterativă.

Start ColorareHarti
  citeşte(n)
  citeşte($((a(i, j), j = 1, n), i = 1, n)$)
  pentru $i = 1, n$
    $culoare(i) = 0$
  sfârşit pentru
  $numarTara = 1$
  cât timp $numarTara > 0$
    dacă $numarTara = n + 1$ *atunci*
      scrie($(culoare(i), i = 1, n)$)
      $numarTara = numarTara - 1$
    sfârşit dacă
    dacă $culoare(numarTara) < 4$ *atunci*

```
        culoare(numarTara) = culoare(numarTara) + 1
        dacă BineColorat(a, numarTara, culoare) = adev atunci
          numarTara = numarTara + 1
        sfârşit dacă
      altfel
        culoare(numarTara) = 0
        numarTara = numarTara − 1
      sfârşit dacă
    sfârşit cât timp
  Stop
```

Subalgoritmul *BineColorat(vecin, numarTara, culoare)* returnează adevărat dacă culoarea ţării curente ($numarTara$)satisface restricţiile cerute faţă de ţările anterior colorate.

```
BineColorat(a, numarTara, culoare)
corect = adevarat
i = 1
cât timp corect = adevarat şi i < numarTara
  dacă a(i, numarTara) = 1 şi culoare(i) = culoare(numarTara)
  atunci
    corect = fals
  altfel
    i = i + 1
  sfârşit dacă
sfârşit cât timp
BineColorat = corect
Return
```

(b) Varianta recursivă.

```
Coloreaza(a, n, culoare, numarTara)
dacă numarTara = n + 1 atunci
  scrie( (culoare(i), i = 1, n) )
altfel
  pentru k = 1, 4
    culoare(numarTara) = k
    dacă BineColorat(a, numarTara, culori)= adev atunci
      Coloreaza(a, n, culoare, numarTara + 1)
    sfârşit dacă
  sfârşit pentru
sfârşit dacă
Return
```

Subalgoritmul *BineColorat* este identic cu cel de la varianta iterativă. Utilizarea subalgoritmului recursiv se face astfel:

Start ColorareHarti
  citeşte(n)
  citeşte($((vecin(i,j), j = 1, n), i = 1, n)$)
  Coloreaza($vecin, n, culoare, 1$)
Stop

3. Problema 3.

Vom nota bărbaţii cu $b_1, \ldots, b_n$ iar femeile cu $f_1, \ldots, f_m$. Vom genera delegaţiile folosind doi vectori, unul pentru indicii de bărbaţi, celălalt pentru indicii de femei (vectorii vb, respectiv vf). Fiecare vector are o componenntă de indice 0 care va avea permanent valoarea 0. În vectorul vf vom avea generate p componente ($k \leq p \leq m$), iar în vectorul vb vom avea q elemente ($0 \leq q \leq n$); varianta $q = 0$ există deoarece se pot forma delegaţii în care să nu existe nici un bărbat.

Varianta dată mai jos este recursivă. Pentru subalgoritmul *GenerareDelegatii*, parametrii fc şi bc reprezintă indicele de vector vf, respectiv vb pentru care se efectuează alegerea unei femei, respectiv bărbat.

GenerareDelegatii($p, fc, m, q, bc, n, vf, vb$)
dacă $fc = p + 1$ *atunci*
  {am generat toţi indiciii pentru femei}
  dacă $bc = q + 1$ *atunci*
    {am generat toţi indiciii pentru bărbaţi}
    scrie($(vf(i), i = 1, p)$)
    scrie($(vb(i), i = 1, q)$)
  altfel
    pentru $i = vb(bc - 1) + 1, n - q + bc$
      $vb(bc) = i$
      GenerareDelegatii($p, fc, m, q, bc + 1, n, vf, vb$)
    sfârşit pentru
  sfârşit dacă
altfel
  pentru $i = vf(fc - 1) + 1, m - p + fc$
    $vf(fc) = i$
    GenerareDelegatii($p, fc + 1, m, q, bc, n, vf, vb$)
  sfârşit pentru
sfârşit dacă
Return

Start Delegatii

citeşte(m, n, k)
$vf(0) = vb(0) = 0$
pentru $p = k, m$
pentru $q = 0, n$
GenerareDelegatii($p, 1, m, q, 1, n, vf, vb$)
sfârşit pentru
sfârşit pentru
Stop

Pentru o variantă iterativă, a se citi capitolul 7, sectiunea 7.5 (generare de combinări).

4. Problema 4.

Vom da mai jos o variantă recursivă. Considerăm $rest$ ca fiind restul care mai este de plătit, $solutie$ un vector cu n elemente, $0 \leq solutie(i) \leq b(i)$, bc tipul bancnotei curente.

Plata($rest, n, solutie, bc$)
dacă $rest = 0$ *atunci*
scrie($(solutie(i), i = 1, bc - 1)$)
altfel
dacă $rest > 0$ şi $bc \leq n$ *atunci*
pentru $k = 0, b(bc)$
$solutie(bc) = k$
Plata($rest - k \cdot v(bc), solutie, bc + 1$)
sfârşit pentru
sfârşit dacă
sfârşit dacă
Return
Start PlataSuma
citeşte(n, S)
citeşte($(b(i), i = 1, n)$)
citeşte($(v(i), i = 1, n)$)
Plata($S, n, solutie, 1$)
Stop

Observaţie Orice problemă cu datele ca mai sus poate fi transformată într–una în care vectorul b conţine doar valoarea 1, prin crearea unui vector v' care va conţine valoarea $v(i)$ de $b(i)$ ori.

Pentru o variantă iterativă în cazul $b(i) = 1, \forall i = 1, n$ a se vedea capitolul 7, secţiunea 7.8

B.5 Rezolvări pentru capitolul 7

1. Problema 1.

   (a) Produs cartezian:

   CartezianRecursiv(m, n, k, x)
   dacă $k = n + 1$ *atunci*
   Afisare(x, m){sau orice subalgoritm care prelucrează informaţia obţinută}
   altfel
   pentru $p = 1, n(k)$
   $x(k) = p$
   CartezianRecursiv($m, n, k + 1, x$)
   sfârşit pentru
   sfârşit dacă
   Return

   Subalgoritmul Afisare este:

   Afisare(a, l)
   pentru $i = 1, l$
   scrie($a(i)$)
   sfârşit pentru
   Return

   Programul principal este:

   Start Cartezian
   citeşte($m, (n(i), i = 1, m)$)
   CartezianRecrsiv($m, n, 1, x$)
   Stop

   Subalgoritmul recursiv generează soluţiile în ordine lexicografică.

   (b) Submulţimi: vom folosi un subalgoritm recursiv cu forma: SubmultimiRecursiv($n, k, cate, x$) unde n este dată de intrare reprezentând numărul elementelor mulţimii iniţiale, k este indicele curent pentru care se încearcă generarea, *cate* reprezintă numărul de elemente ale submulţimii care se generează ($1 \leq k \leq cate$, pentru $cate > 0$), iar x este vectorul soluţie. Vom considera că x are un elemente de indice 0 care va avea permanent valoarea 0.

   SumbultimiRecursiv($n, k, cate, x$)
   dacă $k > cate$ *atunci*

ScrieSolutie($x, cate$){sau orice subalgoritm care prelucrează informaţia dată}
altfel
pentru $p = x(k-1), n - cate + k$
$x(k) = p$
SubmultimiRecursiv($n, k+1, cate, x$)
sfârşit pentru
sfârşit dacă
Return

Subalgoritmul *ScrieSolutie* este o variatie a celui anterior, care în plus ia în considerare cazul încare s-a generat mulţimea vidă:

ScrieSolutie(a, l)
dacă $l = 0$ *atunci*
scrie("multimea vida")
altfel
pentru $i = 1, l$
scrie($a(i)$)
sfârşit pentru
sfârşit dacă
Return

Subalgoritmul principal este:
Start Submultimi
citeşte(n)
$x(0) = 0$
pentru $cate = 0, n$
SubmultimiRecursiv($n, 1, cate, x$)
sfârşit pentru
Stop

Subalgoritmul recursiv generează submulţimile în ordine lexicografică.

(c) Combinări: subalgoritmul este acelaşi cu cel de la generarea recursivă a submulţimilor, cu acceeaşi semnificaţie a parametrilor de apel. Programul principal este:
Start Combinari
citeşte(n, m)
$x(0) = 0$
CombinariRecursiv($n, 1, m, x$)
Stop

Subalgorimul recursiv generează combinările în ordine lexicografică.

(d) Permutări:

```
PermutariRecursiv(k, x, n)
dacă k = 0 atunci
  ScrieSolutie(v, n)
altfel
  pentru i = 1, k
    v(k) ↔ v(i)
    PermutariRecursiv(k − 1, x, n)
    v(k) ↔ v(i)
  sfârşit pentru
sfârşit dacă
Return
```

Subalgoritmul de *ScrieSolutie* este cel de la produs cartezian. Programul principal ar fi:

```
Start Permutari
  citeşte( n )
  pentru i = 1, n
    v(i) = i
  sfârşit pentru
  PermutariRecursiv(n, x, n)
Stop
```

Subalgoritmul recursiv nu generează permutările în ordine lexicografică. Sugerăm cititorului scrierea unei variante bazate pe backtracking recursiv, care să genereze permutările în ordine lexicografică.

B.6 Rezolvări pentru capitolul 8

1. Problema 1. Vom proceda în felul următor: dacă trebuie să umplem un vector al cărui capăt stâng este *st* şi cel drept este *dr*, atunci la mijloc se pune perechea (i, j), după care se completează jumătatea stângă a vectorului, urmată de cea dreaptă. Varianta de mai jos este iterativă, folosind o stivă pentru simularea recursivităţii. Soluţiile vor fi memorate într–o matrice cu $2^n - 1$ linii şi 2 coloane.

```
Start Hanoi
  citeşte( n )
  i = 1
```

$j = 2$
$st = 1$
$dr = 2^n - 1$
$S = \emptyset$\{S este stivă\}
$(i, j, st, dr) \Rightarrow S$
cât timp $S \neq \emptyset$
  $(i, j, st, dr) \Leftarrow S$
  $sol(\frac{st+dr}{2}, 1) = i$
  $sol(\frac{st+dr}{2}, 2) = j$
  dacă $st \leq \frac{st+dr}{2} - 1$ *atunci*
    $(i, 6 - i - j, st, \frac{st+dr}{2} - 1) \Rightarrow S$
  sfârşit dacă
  dacă $\frac{st+dr}{2} + 1, dr$ *atunci*
    $(6 - i - j, j, \frac{st+dr}{2} + 1, dr) \Rightarrow S$
  sfârşit dacă
sfârşit cât timp
scrie($(sol(i, 1), sol(1, 2), i = 1, 2^n - 1)$)
Stop

Adâncimea stivei S este $\Theta(\log_2(2^n - 1)) = \Theta(n)$, iar complexitatea este evident $\Theta(2^n)$.

2. Problema 2. A se vedea rezolvarea de la varianta 4b, pagina 138. Complexitatea obţinută este $T(n) = \Theta(\log n)$.

3. Problema 3. A se vedea rezolvarea de la varianta 4a, pagina 137, pentru care se obţinea o complexitate liniară.

4. Problema 4. La fiecare pas din ciclul *repetă* al algoritmului de parţiţionare i creşte cu o unitate, iar j scade cu o unitate. Parţiţionarea se opreşte atunci când j este la jumătatea şirului, deci împărţirea este echilibrată. Ca atare, complexitatea este Dată de $T(n) = T(n/2) + c \cdot n$, de unde $T(n) = \Theta(n \cdot logn)$.

5. Problema 5. Vom crea o funcţie care determină dacă un număr este sau nu situat între alte două numere:

   Intre(x, y, z)\{returnează adevarat dacă x este între y şi z, fals altfel\}
   dacă ($y \leq x$ şi $x \leq z$) sau ($z \leq x$ şi $x \leq y$) *atunci*
     $Intre = adevarat$
   altfel
     Return fals
   sfârşit dacă

Algoritmul de partiţionare modificat este:

Partitie(a, p, q)
$r = \frac{p+q}{2}$
dacă Intre$(a(p, a(q), a(r)))$ *atunci*
  $indiceMediana = p$
sfârşit dacă
dacă Intre$(a(q), a(p), a(r))$ *atunci*
  $indiceMediana = q$
sfârşit dacă
dacă Intre$(a(r), a(p), a(q)$ *atunci*
  $indiceMediana = r$
sfârşit dacă
dacă $p \neq indiceMediana$ *atunci*
  $a(p) \leftrightarrow a(indiceMediana)$
sfârşit dacă
$pivot = a(p)$
$i = p - 1$
$j = r + 1$
$terminat = fals$
repetă
  repetă
    $j = j - 1$
  până când $a(j) \leq pivot$
  repetă
    $i = i + 1$
  până când $a(i) \geq pivot$
  dacă $i < j$ *atunci*
    $a(i) \leftrightarrow a(j)$
  altfel
    $k = j$
    $terminat = adevarat$
  sfârşit dacă
până când $terminat = adevarat$
Return

B.7 Rezolvări pentru capitolul 9

1. Problema 1. Rezolvarea problemei se face în doi paşi:

   (a) Se determină mediana şirului iniţial $x = (x(1), \ldots, x(n))$; fie ea

med_x

(b) Se calculează şirul auxiliar $y = (y(i), i = 1, n)$ unde $y(i) = |x(i) - med_x|$

(c) Se determină a k-a statistică de ordine a şirului y; fie ea med_y

(d) Se determină k elemente mai mici sau egale cu med_y; indicii acestor elemente sunt indicii elementelor căutate din şirul x.

Complexitatea fiecărui pas de mai sus este $\Theta(n)$, deci complexitatea algoritumului schiţat este $\Theta(n)$. Subalgoritmul în pseudocod este:

Start Apropiate
  citeşte($n, k, (x(i), i = 1, n)$)
  Selectie$(x, 1, n, k, med_x)$
  ObtineY(x, n, y, med_x)
  Selectie$(x, 1, n, k, margine_y)$
  Vecini$(x, y, n, margine_y, z)$
Stop

Subalgoritmul Selectie este cel care determină a k-a statistică de ordine a unui şir (cu complexitate liniară pentru cazul cel mai defavorabil); subalgoritmul ObtineY este:

ObtineY(x, n, y, med_x)
pentru $i = 1, n$
  $y(i) = |x(i) - med_x|$
sfârşit pentru
Return

Subalgoritmul Vecini va determina cei mai apropiaţi k vecini ai şirului x faţă de mediana sa:

Vecini$(x, y, n, margine_y, z)$
$lungZ = 0$
pentru $i = 1, n$
  dacă $y(i) \leq margine_y$ *atunci*
    $lungZ = lungZ + 1$
    $z(lungZ) = x(i)$
  sfârşit dacă
sfârşit pentru
Return

2. Problema 2. Vom folosi faptul că cele două şiruri sunt sortate şi au aceeaşi lungime. De asemenea ne folosim de observaţia evidentă că mediana unui şir sortat se determină în timp constant: dacă i este indicele primului element din şir, iar j este indicele ultimului element al şirului, atunci mediana este pe poziţia $k = \lfloor \frac{i+j}{2} \rfloor$.

   Pentru două şiruri $A(i_A \dots j_A)$ şi $B(i_B \dots j_B)$ cu acelaşi număr de elemente ($j_A - i_A = j_B - i_B$) procedăm în felul următor: dacă lungimile lor sunt mai mici sau egale cu 2, atunci le putem interclasa şi obţinem mediana lor (timp $\Theta(1)$). Dacă numărul de elemente este mai mare decât 2, atunci fie k_A respectiv k_B indicii mijloacelor vectorilor $A(i_A \dots j_A)$ şi $B(i_B \dots j_B)$. Dacă $A(k_A) < B(k_B)$ atunci vom ignora elementele din $A(i_A \dots j_A)$ aflate la stânga indicelui k_A şi elementele din $B(i_B \dots j_B)$ aflate la dreapta indicelui k_B şi vom considera doar vectorii $A(k_A \dots j_A)$ şi $B(k_b \dots j_B)$. Analog se procedează dacă $A(k_A) \geq B(k_B)$. Procesul se reia pentru vectorii înjumătăţiti.

   Justificăm faptul că prin eliminarea unei jumătăţi din fiecare vector nu se pierde mediana căutată. Fie de exemplu $A(k_A) < B(k_B)$. Fie $n = j_A - i_A = j_B - i_B$. Fie i astfel încât $i_A \leq i < k_A$. Atunci $A(i) \leq A_(k_A) \leq A_(p)\ \forall p = k_A, \dots j_A$, $A(i) \leq A(k_A) < B(k_B) \leq B(q)\ \forall q = k_B, \dots j_B$. Obţinem că $A(i)$ este mai mic sau egal decât $2\left(n - \lfloor \frac{n+1}{2} \rfloor + 1\right)$ elemente. Dar $2\left(n - \lfloor \frac{n+1}{2} \rfloor + 1\right) \geq n + 1$, deci $A(i)$ este mai mic sau egal decât cel puţin $n + 1$ elemente. Ca atare, ignorându–le, nu pierdem mediana. Analog se demonstrează că eliminând jumătatea dreaptă a lui B nu se pierde mediana. Demonstraţia este asemănătoare pentru cazul în care $A(k_A) \geq B(k_B)$.

   Algoritmul este dat mai jos:

```
Start MedianaSiruri
  citeşte( n, (A(i), B(i), i = 1, n) )
  i_A = i_B = 1
  j_A = j_B = n
  cât timp i_A < i_B
    k_A = (i_A + j_A)/2
    k_B = (i_B + j_B)/2
    n = n/2
    dacă A(k_A) < B(k_B) atunci
      i_A = k_A
      j_B = k_B
    altfel
      j_A = k_A
```

$i_B = k_B$
sfârşit dacă
sfârşit cât timp
scrie($\max(A(i_A), B(i_B))$)
Stop

unde max returnează maximul a două numere. Complexitatea este:

$$T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$$

3. Problema 3. Se poate proceda în mai multe moduri:

   (a) Sortăm numerele şi alegem cele mai mari i numere

   (b) Construim un (max–)heap şi extragem din el de i ori maximul

   (c) Prin utilizarea unui algoritm de statistică de ordine

   Lăsăm în seama cititorului scrierea algoritmulor pentru cazurile 3a şi 3b. Pentru situaţia 3c determinăm a $n - i + 1$–a statistică de ordine (fie ea x) şi apoi printr–o parcurgere a şirului numerele mai mari sau egale cu x. Complexitatea este de $\Theta(n + i \cdot \log i)$. Sugerăm cititorului compararea acestei complexităţi cu cea obţinută la punctele 3a şi 3b.

4. Problema 4, pagina 4. Ideea este de a "deplasa" elementele de–a lungul axei astfel încât să devină toate pozitive (caz pentru care algoritmul Countsort funcţionează). Această deplasare se obţine adunând o cantitate potrivită, care poate fi luată ca $-\min_{i=1,n}\{x(i)\}$!. După ce se efectuează sortarea, se revine la valorile originale prin scăderea cantităţii adunate anterior.

   Start CountSortModificat
   citeşte($n, (x(i), i = 1, n)$)
   $min = MinimSir(x, n)$ {Functie care determina minimul unui sir}
   pentru $i = 1, n$
   $x(i) = x(i) + (-min)$
   sfârşit pentru
   {Aici apare algoritmul CountSort}
   pentru $i = 1, n$
   $y(i) = y(i) - (-min)$
   sfârşit pentru
   Stop

B.8 Rezolvări pentru capitolul 10

1. Problema 1. Pentru fiecare element $x(i)$ se determină dacă apare şi în şirul y. În caz afirmativ, $x(i)$ va face parte şi din mulţimea intersecţie.

   Faptul că în acest fel se obţine mulţimea intersecţie este (sperăm) evident. Complexitatea algoritmului este dependentă de modul în care se face căutarea lui $x(i)$ în y.

   - Dacă se face căutare liniară (vezi rezolvarea 3, pagina 136), atunci complexitatea este $\Theta(mn)$
   - Dacă se cauta binar (vezi rezolvarea de la 4b, pagina 138), atunci:
     - y trebuie sortat în prealabil, deci avem o complexitate $\Theta(m \log_2 m)$
     - $x(i)$ este căutat binar în y, pentru $i = 1, n$, deci se mai adaugă complexitatea $\Theta(n \log_2 m)$

     În total se obţine o complexitate de $\Theta((m+n)\log m)$. Este mai avantajos ca să se facă sortarea şi căutarea în şirul cel mai scurt.

   Lăsăm cititorul să conceapă algotritmul pentru rezolvarea acestei probleme. În ceea ce priveşte funcţia care se optimizează, este vorba de:

   $$\max f(x, y) = |\{a : a \text{ în } x \text{ şi } a \text{ în } y\}|$$

   ($|A|$ reprezintă numărul de elemente ale mulţimii A), care se regăşeste şi în algoritmul prezentat (cu toate că nu este evident).

2. Problema 2. Ideea algoritmului este de a calcula pentru fiecare obiect valoarea lui unitară, *i.e.* cât se câştigă din transportarea unei unităţi din obiectul respectiv. Suntem mai interesaţi în a considera obiectele profitabile (preţ mare, greutate mică) înaintea celor pentru care raportul preţ/greutate este mic. Algoritmul dat mai jos sintetizează această strategie.

   Start RucsacContinuu
   citeşte($n, G, (c(i), g(i), i = 1, n)$)
   pentru $i = 1, n$
   $v(i) = \frac{c(i)}{g(i)}$
   sfârşit pentru
   Sortare(c, g, v, n)
   $gc = 0$ {gc reprezintă greutatea curentă pusă în rucsac}

$k = 1\{k$ este numărul obiectului curent care se încearcă a fi pu în rucsac$\}$
$S = 0\{S$ este suma realizată prin luarea obiectelor$\}$
cât timp $k \geq n$ şi $gc < G$
  dacă $gc + g(k) \geq G$ *atunci*
    $f(k) = 1\{f(k)$ reprezintă fracţiunea care se ia din obiectul $k\}$
    $gc = gc + g(k)$
    $S = S + c(k)$
    $k = k + 1$
  altfel
    $f(k) = \frac{G-gc}{g(k)}$
    $gc = G$
    $S = S + f(k)g(k)$
  sfârşit dacă
sfârşit cât timp
dacă $k > n$ *atunci*
  $k = n\{$În cazul în care G este mai mare sau egal cu greutatea tuturor obiectelor$\}$
sfârşit dacă$\{$
pentru $\}$
  $i = 1, k$
  scrie$(\ f(i)\ )$
sfârşit pentru
Stop

Subalgoritmul *Sortare*(c, g, v, n) rearanjează elementele vectorilor c, g, v astfel încât $v(1) \geq v(2) \geq \cdots \geq v(n)$. Corectitudinea algoritmului de mai sus este dată de propoziţia de mai jos:

Propozitia 1. *Algoritmul precedent determină câştigul maxim în restricţiile impuse problemei.*

Demonstraţie. Dacă $G > \sum_{i=1}^{n} g(i)$, atunci în mod evident soluţia dată de vectorul f va avea forma $f = (1, 1, \ldots, 1)$ (se iau toate obiectele), ceea ce evident înseamnă optim. Vom considera deci cazul în care $G < \sum g(i)$. Este uşor de văzut că soluţia dată de vectorul f are forma:

$$f = (1, 1, \ldots, 1, \epsilon, 0, 0, \ldots 0) \tag{B.1}$$

unde $0 < \epsilon \leq 1$ şi ϵ apare pe poziţia k (k fiind variabila din algoritm).

Avem că:

$$S = \sum_{i=1}^{n} f(i)c(i) \tag{B.2}$$

şi

$$G = \sum_{i=1}^{k} f(i)g(i) \tag{B.3}$$

Fie S' câştigul maxim care se poate realiza în condiţiile problemei şi fie

$$f' = (f'(1), \ldots, f'(n)) \tag{B.4}$$

fracţiunile din obiecte care se iau, pentru a realiza S':

$$S' = \sum_{i=1}^{n} f'(i)c(i) \tag{B.5}$$

Avem:

$$G' = \sum_{i=1}^{n} f'(i)g(i) \tag{B.6}$$

unde G' este greutatea totală care se transportă pentru câţigul S'.

Putem avea următoarele situaţii:

(a) Dacă (prin absurd) $S' < S$, atunci se contrazice optimalitatea lui S'; deci această situaţie nu poate să apară.

(b) Dacă $S' = S$, atunci S este optimal şi nu mai avem ce demonstra.

(c) Dacă (prin absurd) $S' > S$, atunci procedăm după cum urmează. Fie p cel mai mic indice pentru care $f'(p) \neq 1$ (în ipoteza în care un asemenea p există). Vom transforma $f'(p)$ în $f''(p) = f'(p) + \alpha = 1$, obţinând un surplus de greutate de $\alpha g(p)$ (depăşim greutatea maximă G). Acest surplus de greutate va fi scăzut dintr–un alt obiect de indice t, $t > p$ (sau din mai multe obiecte de indici mai mari decât p, dar tratarea este analoagă cu cea pentru un singur obiect). Vom schimba deci $f'(t) \rightarrow f''(t) = f'(t) - \beta \geq 0$, astfel încât

$$\beta g(t) = \alpha g(p) \tag{B.7}$$

Pentru $i \neq p, t$ vom avea $f''(i) = f'(i)$. Atunci:

$$\begin{aligned} S'' &= \sum_{i} f''(i)c(i) = \sum_{i \neq p,t} f'(i)c(i) + (f'(p) + \alpha)c(p) + \\ &\quad (f'(t) - \beta)c(t) = S' + \alpha c(p) - \beta c(t) \end{aligned} \tag{B.8}$$

Dar: $\alpha c(p) - \beta c(t) = \alpha v(p)g(p) - \beta v(t)g(t)$ şi conform relaţiei (B.7) obţinem că:

$$S'' = S' + \alpha g(p)(v(p) - v(t)) \tag{B.9}$$

Dacă $v(p) - v(t) = 0$, atunci $S' = S''$ şi procedeul se continuă cu următorul p. Dacă $S' < S''$ se contrazice optimalitatea lui S'.

Dacă nu (mai) există nici un p cu proprietatea $f'(p) < f(p) = 1$, mai avem de studiat relaţia dintre $f(k) = \epsilon$ şi $f'(k)$. Dacă $f'(k) < \epsilon$, atunci printr–un procedeu analog de mărire a lui $f'(k)$ la ϵ (şi scăderea unui $f'(t)$, $t > k$) vom obţine fie o contradicţie, fie $S'' = S'$ (ca mai sus). Dacă $f'(k) > \epsilon$, atunci se contrazice modul de alegere al lui $f'(k) = \epsilon$ din algoritm. În sfârşit, dacă $f'(k) = \epsilon$, atunci avem că f' poate fi adus la forma f fără a pierde din câştig.

Deci am demonstrat că S dat de f (conform algoritmului) nu poate fi mai mic decât S', de unde rezultă propoziţia. □

3. Problema 3. Algoritmul sugerat în [4] este:

   (a) Se sortează spectacolele după ora terminării lor.

   (b) Primul spectacol programat este cel care începe cel mai devreme.

   (c) Se alege primul spectacol din cele ce urmează în şir ultimului spectacol programat care îndeplineşte conmdiţia că începe după ce s–a terminat ultimul spectacol programat.

   (d) dacă tentativa de mai sus a eşuat (nu am găsit un astfel de spectacol), algoritmul se termină, altfel se programează spectacolul găsit şi se reia pasul 3c.

   Demonstraţia faptului că acest algoritm duce la soluţia optimă este dată în [4].

   Algoritmul este:

   Start Spectacole
   citeşte($n, (st(i), sf(i), i = 1, n)$)
   Sortare(st, sf, n)
   scrie("Spectacol care începe la ", $st(1)$, "şi se sfârşeşte la", $sf(1)$)

$k = 1$
pentru $i = 2, n$
  dacă $st(i) \geq sf(k)$ *atunci*
    scrie("Spectacol care începe la ", $st(i)$, "şi se sfârşeşte la", $sf(i)$)
    $k = i$
  sfârşit dacă
sfârşit pentru
Stop

Algoritmul $Sortarest, sf, n$ interschimbă elementele din st, respectiv sf astfel încât $sf(1) \leq sf(2) \leq \cdots \leq sf(n)$.

Complexitate: avem o sortare care se poate face în timp $\Theta(n \log n)$ şi o parcurgere a şirului st, deci în total $\Theta(n \log n)$.

4. Problema 4. Deşi problema are un caracter teoretic pronunţat, rezultatul obţinut poate fi folosit în numeroase cazuri. În [4] se arată că strategia pentru această problemă este:

   (a) Se sortează elementele celor două mulţimi

   (b) Dacă $A = \{a_1, \ldots, a_m\}$ are k elemente negative şi p elemente pozitive, se aleg primele k elemente ale lui B şi ultimele p elemente ale lui B.

   Algoritmul este:
   Start Maxim
   citeşte($m, n, (a(i), i = 1, m), (b(i), i = 1, n)$)
   Sortare(a, m)
   Sortare(b, n)
   $k = 1$
   cât timp $k \leq m$ şi $a(k) < 0$
     $k = k + 1$
   sfârşit cât timp
   $k = k - 1$
   $suma = 0$
   pentru $i = 1, k$
     $suma = suma + a(i) * b(i)$
   sfârşit pentru
   pentru $i = k + 1, m$
     $suma = suma + a(i) * b(n - m + i)$

sfârşit pentru
Stop

5. Problema 5. Timpul de aşteptare pentru un client este compus din timpii de servire ai clienţilor care sunt procesaţi înaintea lui adunat cu timpul de servire pentru el însuşi. De aceea timpul total de aşteptare nu este indiferent faţă de ordinea de servire. De exemplu, dacă avem doi clienţi, cu $t_1 = 1$ şi $t_2 = 2$, atunci servirea în ordinea $1, 2$ duce la un timp de aşteptare de $1 + (1 + 2) = 4$, pe cˆdacă ar fi procesaţi invers, s–ar ajunge la timpul $2 + (2 + 1) = 5$. Intuitiv, este mai bine dacă clienţii sunt procesaţi în ordinea vrescătoare a timpilor de servire (în acest fel un timp de servire mare are un impact mai târziu).

Fie p o permutare a indicilor clienţilor (corespunzând unei ordini de intrare); clienţii sunt preluaţi în ordinea $p_1, p_2, \ldots, p_n$. Pentru această ordine, timpul total de servire este:

$$T(p) = t_{p_1} + (t_{p_1} + t_{p_2}) + \cdots + (t_{p_1} + t_{p_2} + \cdots + t_{p_n}) = \sum_{i=1}^{n} (n - i + 1) t_{p_k} \tag{B.10}$$

Fie p permutarea care determină T din (B.10) minim. Fie p' permutarea care se obţine din p printr–o inversiune: $p'(i) = p(j)$, $p'(j) = p(i)$ $(i < j)$, $p'(k) = p(k), \forall k \neq i, j$. Atunci avem că $T(p) \leq T(p')$, ceea ce este echivalent cu $(j - i)(t_{p_i} - t_{p_j}) \leq 0$, ceea ce conduce la $t_{p_i} < t_{p_j}$, ceea ce confirmă valabilitatea strategiei.

Algoritmul este:

Start Procesare
  citeşte($n, (t(i), i = 1, n)$)
  pentru $i = 1, n$
    $p(i) = i$
  sfârşit pentru
  Sortaret, p, n
  pentru $i = 1, n$
    scrie($p(i)$)
  sfârşit pentru
Stop

Algoritmul *Sortare*(t, p, n) sortează elementele lui t, în paralel făcând aceleaşi interschimbări în vectorul de permutare p.

B.9 Rezolvări pentru capitolul 11

1. Problema 1. Tablourile m şi s, analoage celor din Tabelul 11.2, pagina 116, au valorile:

1	2	3	4	5	6	1	2	3	4	5	6
0	150	330	405	1655	2010	0	1	2	2	4	2
0	0	360	330	2430	1950	0	0	2	2	2	2
0	0	0	180	930	1770	0	0	0	3	4	4
0	0	0	0	3000	1860	0	0	0	0	4	4
0	0	0	0	0	1500	0	0	0	0	0	5
0	0	0	0	0	0	0	0	0	0	0	0

   De aici tragem concluzia că numărul optim de înmulţiri scalare este $m(1,6) = 2010$. Pentru determinarea modului în care se face înmulţirea, folosim informaţia din matricea s: $s(1,6) = 2$, deci vom avea parantezarea $(A_1 \cdot A_2) \cdot (A_3 \cdot \ldots A_6)$. Pentru a determina cum se face parantezarea pentru produsul $A_{3\cdot 6}$, folosim $s(3,6) = 4$, de unde deducem că grupăm $A_{3\ldots 6} = (A_3 \cdot A_4) \cdot (A_5 \cdot A_6)$. În final, obţinem parantezarea completă:

$$((A_1 \cdot A_2) \cdot ((A_3 \cdot A_4) \cdot (A_5 \cdot A_6)))$$

2. Problema 2. Demonstraţia se face prin inducţie matematică după numărul n de matrici. Fie $P(n)$ propoziţia "parantezarea completă a unui şir cu n matrici are exact $n-1$ paranteze".

   Verificarea:

   (a) $P(1)$: o matrice reprezintă o expresie complet parantezată

   (b) $P(2)$: $(A_1 \cdot A_2)$ este singura expresie complet parantezată asociată produsului a două matrici, deci avem o pereche de paranteze.

   Demonstraţie: presupunem propoziţia adevărată până la $n-1$ şi demonstrăm că şi $P(n)$ este adevărată. Fie $A_{1\ldots n} = (A_{1\ldots p} \cdot A_{p+1\ldots n})$, $1 \leq p < n$. Atunci: conform propoziţiei $P(p)$ avem că numărul de perechi de paranteze pentru $A_{1\ldots p}$ este $p-1$, iar pentru $A_{p\ldots n}$ este $n-(p+1)$. Pentru $A_{1\ldots n}$ avem nevoie de $(p-1)+(n-p-1)+1$ perechi de paranteze (ultimul termen din sumă se datorează perechii de paranteze exterioare). Deci $P(n)$ este adevărată, de unde concluzia din enunţul problemei.

(c) Problema 3. Relaţia după care se calculează optimul este:

$$m(i,j) = \min_{i \le k < j} \{m(i,k) + m(k+1,j) + p_{i-1}p_k p_j\} \qquad \text{(B.11)}$$

Fie $\alpha \le \beta$. $m(\alpha,\beta)$ este accesat în cadrul (B.11) fie ca $m(i,k)$, fie ca $m(k+1,j)$. Fie $t(\alpha,\beta)$ numărul de apeluri ce se face pentru $m(\alpha,\beta)$ în primul caz şi $T(\alpha,\beta)$ numărul de apeluri ce rezultă din al doilea caz.

Pentru primul caz, avem $i = \alpha, k = \beta$. Putem avea $j \in \{\beta+1, n\}$, deci $T(\alpha,\beta) = n - \beta$. Pentru $U(\alpha,\beta)$ se poate răta analog că $U(\alpha,\beta) = n - \beta$. Obţinem:

$$R(\alpha,\beta) = T(\alpha,\beta) = \sum_{\alpha=1}^{n}\sum_{\beta=\alpha}^{n}(n - \beta + \alpha - 1) = \cdots = \frac{n^3 - n}{3}$$

(d) Problema 4. Fie $a = (a(i,j))$ matricea triunghiulară dată la intrare ($1 \le i \le n$, $1 \le j \le i$). Observăm că avem 2^{n-1} posibilităţi de a forma drumuri de la $a(1,1)$ la ultima linie (demonstraţia se face prin inducţie matematică după n). Ca atare, o metodă de abordare de tip *brute force* este de la început sortită eşecului.

De asemenea se observă că o strategie de tipul "coboară la numărul cel mai mare aflat pe linia imediat următoare" nu duce la soluţie optimă. Vom da un algoritm bazat pe programare dinamică.

Fie $\mathcal{D}$ drumul de la $a(1,1)$ la ultima linie care dă suma maximă. Fie $S(i,j)$ un element de pe acest drum, aflat pe linia i şi coloana j. Atunci $S(i,j)$ este valoarea maximă care se poate obţine plecând din (i,j) până la ultima linie. Dacă nu ar fi aşa, atunci înseamnă că ar exista un alt drum de la (i,j) la ultima linie care dă valoare mai mare, iar prin modificarea lui $\mathcal{D}$ astfel încât să includă acest ultim drum am obţine o valoare mai mare, contradicţie cu alegerea lui $\mathcal{D}$. Ca atare, putem deduce că:

$$S(i,j) = \begin{cases} S(n,j) & \text{dacă } i = n \\ a(i,j) + \max\{S(i+1,j), S(i+1,j+1)\} & \text{dacă } i < n \end{cases} \qquad \text{(B.12)}$$

pentru $1 \le i \le n, 1 \le j \le i$; ceea ce ne duce la algoritm. Calculul lui $S(i,j)$ nu se face recursiv, ci folosind o matrice triunghiulară. Pentru reconstituirea soluţiei se foloseşte o matrice triunghilară *sol*

Start Triunghi

```
citeşte( n, (a(i, j), i = 1, n, j = 1, i) )
pentru j = 1, n
   s(n, j) = a(n, j)
sfârşit pentru
pentru i = n − 1, 1, −1
   pentru j = 1, i
      dacă s(i + 1, j) > s(i + 1, j + 1) atunci
         s(i, j) = a(i, j) + s(i + 1, j)
         sol(i, j) = j
      altfel
         s(i, j) = a(i, j) + s(i + 1, j)
         sol(i, j) = j + 1
      sfârşit dacă
   sfârşit pentru
sfârşit pentru
scrie( "Suma maximă este", s(1, 1) )
scrie( "Ea se obţine astfel:" )
i = j = 1
scrie( a(1, 1) )
pentru k = 1, n − 1
   j = sol(i, j)
   i = i + 1
   scrie( a(i, j) )
sfârşit pentru
Stop
```

3. Problema 5. Sugerăm cititorului să urmărească rezolvarea problemei 1. Rezolvarea se bazează pe un algoritm de tip *Divide et Impera* din cauză că pentru un produs de tipul $A_{i...j}$ vom determina indicele k pentru care facem descompunerea $A_{i...j} = (A_{i...k}) \cdot (A_{k+1...j})$, fiecare din cele două matrici fiind la rândul lor parantezate independent, până când se ajunge la câte o singură matrice.

```
Descompunere(s.i, j)
dacă i = j atunci
   scrie( i )
   Return
sfârşit dacă
dacă j − i = 1 atunci
   scrie( "(", i,"*", j, ")" )
   Return
sfârşit dacă
```

$k = s(i, j)$
scrie(“(”)
Inmultiri(s, i, k)
scrie(“)*(”)
Inmultiri$(s, k + 1, j)$
scrie(“)”)
Return

Propunem cititorului interesat crearea unui algoritm care să determine *toate* modalităţile de înmulţire a matricelor astfel încât să se ajungă la un număr de înmulţiri scalare minim. (Indicaţie: nu se mai foloseşte informaţiua din s, ci doar cea din m)

Bibliografie

[1] Andonie, Răzvan, and Gârbacea, Ilie. *Algoritmi fundamentali. O perspectivă C++*. Editura Libris, 1995. Download de la: http://vega.unitbv.ro/~andonie.

[2] Cormen, Thomas H, Leiserson, E Charles, and Rivest, Ronald R. *Introducere în algoritmi*. Computer Libris Agora, 2000.

[3] Livovschi, Leon and Georgescu, Horia. *Sinteza şi analiza algoritmilor*. Editura Ştiinţifică şi Enciclopedică, 1986.

[4] Tudor, Sorin. *Tehnici de programare (Manual pentru clasa a X-a)*. L&S Infomat, 1996.