



UNIVERSITATEA *TRANSILVANIA* DIN BRAȘOV

Centrul de Învățământ la Distanță  
și Învățământ cu Frecvență Redusă



---

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ  
PROGRAM DE LICENȚĂ: INFORMATICĂ

# **Algoritmică**

**CURS PENTRU ÎNVĂȚĂMÂNT LA DISTANȚĂ**

**AUTORI**

**Conf. Dr. Paul Iacob**

**ANUL I**

**2012-2013**

---

## Introducere

---

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

1. se detectează problema;
2. se adaugă restricțiile 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 interpretează rezultatele de la calculator în termeni specifici domeniului problemei.

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

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

Originea cuvântului “algoritm” este legată de numele Abu Ja’far Mohammed IbnM<sup>u</sup>sâ al Khowârizmî, autor arab al unei cărți de matematică.

Proprietățile pe care trebuie să le aibă un algoritm sunt:

1. Fiecare pas trebuie să conțină o operație bine definită;
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.
4. Trebuie să se termine în timp finit. că viitoarele mașini 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 și abia apoi a mașinii.
2. Exprimarea noastră va fi restrânsă, din necesitatea de a scrie ușor programe corecte, la algoritmi structurați.
3. Vom învăța să verificăm corectitudinea algoritmilor concepuți.
4. Nu ajunge să construim algoritmi corecți. Trebuie să cunoaștem și performanțele algoritmilor creați.



### Obiectivele cursului

Lucrarea de față își propune să fie un curs introductiv în teoria algoritmilor.

Se vor introduce noțiunile de algoritm, subalgoritm, complexitatea algoritmilor, diferite metode de elaborare a algoritmilor, probleme clasice.

Cursul algoritmică este un curs de bază pentru meseria de informatician.

La sfârșitul acestui curs, studenții vor fi capabili să:

- elaboreze algoritmi
- sa analizeze complexitatea algoritmului
- să demonstreze corectitudinea algoritmului



### **Cerințe preliminare**

Cursul se va susține studenților înainte de orice alt curs.

Cunoștințele acumulate la acest curs pot fi utile la înțelegerea problemelor ridicate de compilatoare și la elaborarea programelor în diverse limbaje de programare.



### **Resurse**

Pentru a înțelege algoritmi proiectați și pentru a vedea cum lucrează se va folosi orice limbaj (de exemplu c++ sau Pascal)



### **Structura cursului**

Cursul de algoritmică este structurat în două module, astfel: primul modul cuprinde patru unități de învățare, iar al doilea modul cuprinde șase unități de învățare. La rândul său, fiecare unitate de învățare cuprinde: obiective, aspecte teoretice privind tematica unității de învățare respective, exemple, teste de autoevaluare precum și probleme propuse spre discuție și rezolvare.

La sfârșitul fiecărui modul sunt date teste de autoevaluare. Rezolvarea acestor teste se găsește la sfârșitul cursului.



### **Durata medie de studiu individual**

Parcurgerea de către studenți a unităților de învățare ale cursului de baze de date (atât aspectele teoretice cât și rezolvarea testelor de autoevaluare și rezolvarea problemelor propuse) se poate face în 2-3 ore pentru fiecare unitate.



### **Evaluarea**

La sfârșitul semestrului, fiecare student va primi o notă, care va cuprinde: un examen scris cu materia din modulele care va deține o pondere de 60% în nota finală și notele aferente algoritmilor elaborați la seminar și programelor elaborate la laborator.

**Spor la treabă**

## CUPRINS

Modulul 1. Limbajul pseudocod și structuri elementare.....	4
M1.U1. Istoric; comentarii.....	4
M1.U2. Limbajul de descriere al algoritmilor: pseudocod. ....	7
M1.U3. Analiza algoritmilor.....	12
M1.U4. Structuri elementare.....	20
M1.U5.Recursivitate.. ....	29
Modulul 2. Metode de elaborare a algoritmilor. ....	36
M2.U1. Metoda Backtracking.....	36
M2.U2. Generarea submulțimilor .....	41
M2.U3. Metoda Divide et Impera .....	53
M2.U4. Sortare și statistici de ordine.....	61
M2.U5. Metoda Greedy .....	71
M2.U6. Metoda programării dinamice.....	81
Răspunsuri la teste.....	90
M1.U2.3. Test de autoevaluare a cunoștințelor.....	90
M1.U4.3. Test de autoevaluare a cunoștințelor.....	96
M1.U5.3. Test de autoevaluare a cunoștințelor.....	102
M2.U1.3. Test de autoevaluare a cunoștințelor.....	104
M2.U2.3. Test de autoevaluare a cunoștințelor.....	111
M2.U3.3. Test de autoevaluare a cunoștințelor.....	113
M2.U4.3. Test de autoevaluare a cunoștințelor.....	116
M2.U5.3. Test de autoevaluare a cunoștințelor.....	118
M2.U6.3. Test de autoevaluare a cunoștințelor.....	123
Bibliografie.....	127

# Modulul 1. Limbajul pseudocod și structuri elementare

---

## Cuprins

Introducere .....	3
U1. Istoric, definiții, generalități .....	4
U2. Limbajul pseudocod .....	7
U3. Analiza algoritmilor .....	12
U4. Structuri elementare .....	20
U5. Recursivitate.....	28



### Introducere.

De la modelul matematic nu se poate, în general, trece direct la program în orice limbaj ar fi el. Este nevoie de un limbaj intermediar care să pună în evidență structuri specifice care să fie, pe urmă, transpuse într-un limbaj de programare. Metoda a parcurs un drum lung, din antichitate și a devenit ceea ce este doar în legătură cu apariția calculatorului.

## M1.U1. Istoric; comentarii



### M1.U1.1. Introducere

Este important să știm cum a evoluat noțiunea și modul de reprezentare a algoritmilor.



### M1.U1.2. Obiectivele unității de învățare

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- Înțeleagă modul cum a evoluat noțiunea de algoritm
- Să înțeleagă că trebuie totdeauna să scriem mai întâi algoritmul și abia apoi programul.



**Durata medie de parcurgere a acestei unități de învățare este de o oră.**

### **Istoric; comentarii**

Abū Abdallah Muhammad ibn Musa al-Khwārizmī, angajat al Casei Întelepciunii din Bagdad, a fost un savant persan, (780 - 850). El a tradus cărțile lui Euclid despre introducerea sistemului de numerație pozițional și rezolvarea algebrică a ecuațiilor de gradul I și II fapt care a adus o importantă contribuție la dezvoltarea matematicii. Opera lui al-Khwārizmī a ajuns în Europa prin intermediul lui Fibonacci. Numele său a suferit numeroase transformări, fie prin latinizare, fie prin omisiuni și din Al-Khwarizmi s-a ajuns la Algorism/Algoritm termen folosit pentru referirea la o serie determinată de pași urmați pentru rezolvarea unei probleme, iar numele al-jabr dat unei operații folosite în cadrul algoritmului de rezolvare a ecuațiilor, a devenit Algebra.

Dezvoltarea științei de a scrie un algoritm este strâns legată de dezvoltarea calculatoarelor și de aplicabilitatea lor în cele mai diverse domenii.

Calculatoarele pot fi folosite pentru a rezolva probleme, numai cu ajutorul unor programe corespunzătoare de rezolvare. Noțiunea de program (programare) a suferit schimbări în istoria informaticii. La început problemele rezolvate cu ajutorul calculatorului erau simple și algoritmii, pentru rezolvarea lor, nu erau prea complicați. Prin program se înțelegea rezultatul scrierii unui algoritm într-un limbaj de programare. Din cauza creșterii complexității problemelor, astăzi adesea se concep pachete de mai multe programe pentru rezolvarea unei probleme.

O definiție matematică, riguroasă, pentru noțiunea de algoritm, este greu de dat, pentru că este o noțiune primară (nu există genul proxim). Descriem, în continuare, ce se înțelege prin algoritm. Ne familiarizăm cu această noțiune prezentând mai multe exemple de algoritmi și observând ce au ei în comun. Cel mai vechi exemplu este algoritmul lui Euclid, algoritm care determină cel mai mare divizor comun a două numere naturale.

Algoritmul reprezintă un pas intermediar între modelul matematic și programul realizat într-un limbaj conceput pentru calculator.

Proprietățile pe care trebuie să le aibă un algoritm sunt:

1. Fiecare pas trebuie să conțină o operație bine definită (de exemplu următoarele operații 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: operații cu numere întregi - da, dar operații 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 și totuși vor fi considerați algoritmi care pe mașinile actuale necesită zeci de ani pentru că se presupune că viitoarele mașini 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 și abia apoi a mașinii. 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 ușor programe corecte, la algoritmi structurați, adică algoritmi care folosesc anumite structuri specifice care vor fi expuse la începutul cursului.
3. Vom învăța să verificăm corectitudinea algoritmilor concepuți. Un algoritm corect trebuie să rezolve problema propusă pentru orice date de intrare. Nu se pune problema verificării cu anumite date (acest lucru ține de verificarea programului), ci problema unei demonstrații a corectitudinii programului (în general ca o teoremă).
4. Nu ajunge să construim algoritmi corecți ci trebuie să cunoaștem și performanțele algoritmilor creați. Pot exista mai mulți algoritmi care să rezolve aceeași problemă, unii mai greu sau mai ușor de înțeles (de urmărit), unii mai lungi în numărul de pași descriși, dar cu performanță urmărită prin *analiză* (numărul de pași elementari executați, în legătură cu dimensiunea intrării). O altă problemă este cea legată de dimensiunea memoriei necesare execuției programului realizat după un anumit algoritm.



### Rezumat

Originea cuvântului “algoritm” este legată de numele Abu Ja’far Mohammed IbnM<sup>^</sup>usâ al Khowârizmî, autor arab al unei cărți de matematică.

Un algoritm trebuie să aibă următoarele proprietăți:

1. Fiecare pas trebuie să conțină o operație bine definită;
2. Pasul care urmează fiecărui pas trebuie să fie unic definit;
3. Fiecare pas trebuie să fie efectuabil.
4. Trebuie să se termine în timp finit.

## ***M1.U2. Limbajul de descriere al algoritmilor: pseudocod.***



### **M1.U2.1. Introducere**

De la modelul matematic construim algoritmul pentru care vom folosi limbajul pseudocod.



### **M1.U2.2. Obiectivele unității de învățare**

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- Să utilizeze structurile de bază ale pseudocodului.
- Să scrie algoritmi simpli.



**Durata medie de parcurgere a acestei unități de învățare este de 3 ore.**

Există două limbaje de descriere a algoritmilor: schema logică și pseudocodul. Limbajul *schemei logice* are avantaje de explicație grafică, dar și dezavantaje legate de structurare și traducere, așa că vom folosi *pseudocodul*. Acest limbaj folosește cuvinte cheie (subliniate), cuvinte care definesc variabile, constante și text explicativ (între acolade). Putem distinge următoarea structură generală:

*Start* Numele programului

*citește*( 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=',  $v$ )

Operația cea mai des folosită este *atribuirea*.

Instrucțiunea  $a=b+c$  trebuie înțeleasă ca:

- ✓  $b$  și  $c$  au valori calculate dinainte sau citite ca date de intrare.
- ✓ valorile lui  $b$  și  $c$  se adună în procesor și rezultatul se va afla într-o memorie a procesorului
- ✓ din această memorie , rezultatul adunării se duce în  $a$ .

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



dacă condiție logică

atunci (grup de pași 1)

altfel (grup de pași 2)

sfârșit dacă



### Exemple

#### Exemplul 1.

Start Algoritm 1

Citește(  $i, a$  )

dacă  $i \leq 1$

atunci  $a = a + i$

altfel  $a = a - i$

sfârșit dacă

scrie( 'a=',  $a$  )

Stop

Acest algoritm scrie  $a = 10$  dacă intrarea este (0, 10), iar pentru intrarea (2, 10) scrie  $a = 8$ .

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

Start Algoritm 2

citește(  $a, b, c$  )

$max = a$

dacă  $max < b$  atunci

$max = b$

sfârșit dacă

dacă  $max < c$  atunci

$max = c$

sfârșit dacă

scrie( 'Maximul este ',  $max$  )

Stop



Urmăriți algoritmul Algoritm 2 pentru valorile de intrare (10, 1, 3), (2, 2, 5), (1, 12, 2).

Există grupuri de instrucțiuni care se execută în mod repetat. Aceste structuri se numesc cicluri și apar des în elaborarea algoritmilor. Prima formă de ciclare, pe care o studiem, se numește ciclu cu numărător și se folosește atunci când se știe de câte ori se execută un grup de instrucțiuni. Are forma:

pentru  $ind = vi, vf, pas$

grup de pași

sfârșit pentru

Dacă  $pas = 1$ , atunci el se poate omite. Grupul de pași 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 pași  
sfârșit pentru



### Exemple

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

Start Algoritm 3

citește(  $n, (a(i), i = 1, n)$  )

$s = 0$  {0 este element neutru pentru adunare}

pentru  $i = 1, n, 1$

$s = s + a(i)$

sfârșit pentru

scrie( 'suma este ',  $s$  )

Stop

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

Start Algoritm 4

citește(  $n, (a(i), i = 1, n)$  )

$np = 0$

$nn = 0$

$nz = 0$

pentru  $i = 1, n$

dacă  $a(i) > 0$  atunci

$np = np + 1$

altfel

dacă  $a(i) < 0$  atunci

$nn = nn + 1$

altfel

$nz = nz + 1$

sfârșit dacă

sfârșit dacă

sfârșit pentru

scrie( 'Pozitive= ',  $np$ , 'Negative= ',  $nn$ , 'Nule= ',  $nz$  )

Stop



Scrieți un algoritm care face produsul numerelor diferite de 0 dintr-un vector cu  $n$  componente ( $A(i), i=1, n$ )

Acest tip de ciclu se numește ciclu cu numărător. Mai avem și cicluri cu condiție, care poate fi înainte sau după corpul ciclului (grupul de pași ce se execută de fiecare dată). Ciclul cu test anterior are forma:

cât timp (condiție logică)

grup de pași

sfârșit cât timp

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



### Exemple

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

Start Euclid

citește(  $a, b$  )

$m = a$

$n = b$

cât timp  $n \neq 0$

$r = \text{rest}(m/n)$

$m = n$

$n = r$

sfârșit cât timp

scrie( 'CMMDC dintre '  $a$  , ' și '  $b$  , ' este '  $m$  )

Stop



1. Urmăriți algoritmul pentru intrările (10, 3) și (258, 12).

2. Scrieți un algoritm care adună componentele unui vector până la prima componentă negativă.

Ciclul cu test posterior are forma:

repetă

grup de pași

până când condiție logică

Grupul de pași se execută cel puțin o dată.



### Exemple

**Exemplul 6.** Se cere listarea combinațiilor din {1, 2, 3, 4, 5} luate câte 3:

Start Combinații

pentru  $i = 1, 3$

$j = i$

repetă

$j = j + 1$

$k = j$

repetă

$k = k + 1$

scrie( i, ' ', j, ' ', k ) {afișează o combinație}

până când  $k = 5$

până când  $j = 4$

sfârșit pentru

Stop

Organizați execuția algoritmului de mai sus folosind un tabel de gestionare a datelor de intrare și de ieșire.



### Rezumat

Un algoritm are o structură bine definită.

Instrucțiunile prezentate, împreună cu noțiunea de subalgoritm formează un vocabular suficient pentru scrierea algoritmilor.

S-au prezentat: forma generală a unui algoritm, citirea și scrierea datelor, atribuirea, instrucțiunea de decizie și cele trei forme de ciclare.

### M1.U2.3. Test de autoevaluare a cunoștințelor

1) Se definește produsul scalar a doi vectori A și B, de aceeași dimensiune n

cu formula:  $PS = \sum_{i=1}^n A(i) \cdot B(i)$ . Să se scrie algoritmul pentru calculul

produsului scalar.

2). Se dau n, vectorul A(i),  $i=1, n$  și un număr x. Să se scrie un algoritm care găsește indicele j pentru care  $A(j)=x$  (problema căutării).

3. Cum se modifică algoritmul de mai sus dacă știm că vectorul este sortat crescător?

4. Se dau n și vectorul A(i),  $i=1, n$ . Să se găsească cel mai mic element.

5. Reluați problema anterioară în situația în care trebuie să aflați câte elemente minime sunt. Încercați să rezolvați problema folosind o singură parcurgere a șirului.

6. Se dau n și vectorul A(i),  $i=1, n$ . Ce număr apare cel mai des și de câte ori?

7. Cum se modifică algoritmul precedent dacă vectorul este deja sortat ?



## M1.U3. Analiza algoritmilor



### M1.U3.1. Introducere

Putem ușor să ne dăm seama că pentru a rezolva o anumită problemă pot exista mai mulți algoritmi. De exemplu, se știe 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 numerelor în produse de factori primi.

Se pune întrebarea:

Care dintre acești algoritmi este mai bun?

și în general:

Cum putem compara doi algoritmi care rezolvă aceeași problemă?

Cu răspunsul la aceste întrebări se ocupă analiza algoritmilor.



### M1.U3.2. Obiectivele unității de învățare

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- estimeze timpul necesar execuției unui algoritm
- utilizeze corect funcțiile asimptotice.



**Durata medie de parcurgere a acestei unități de învățare este de 2 ore.**

## Analiza algoritmilor

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

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

Pentru problema sortării avem: *la intrare* un șir de  $n$  valori numerice ( $a_1, a_2, \dots, a_n$ ) și vrem *la ieșire* un șir ( $a_{\_1}, a_{\_2}, \dots, a_{\_n}$ ), obținut prin permutarea valorilor inițiale, astfel încât  $a_{\_1} \leq a_{\_2} \leq \dots \leq a_{\_n}$ . Ideile algoritmului descris mai jos (sortare prin interclasare) pot fi înțelese dacă ne gândim la aranjarea cărților de joc: avem pe masă, cu fața în jos, o serie de cărți pe care le putem lua una câte una în mână, deplasându-le până la locul pe care trebuie să îl ocupe.



### Exemplu

Fie șirul  $A = 3\ 1\ 5\ 2\ 7\ 4$ . Acest șir va fi parcurs de un cursor  $i$  (se ia câte o carte) și mutat în cadrul șirului 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 fața lui:  $A_{\_} = 1\ 3$

– pentru  $i = 3$ :  $A_{\_} = 1\ 3\ 5$

- pentru  $i = 4$ :  $A_4 = 1\ 2\ 3\ 5$
- pentru  $i = 5$ :  $A_5 = 1\ 2\ 3\ 5\ 7$
- pentru  $i = 6$ :  $A_6 = 1\ 2\ 3\ 4\ 5\ 7$



Urmăriți, într-un tabel, execuția algoritmului de mai jos cu datele de intrare:  
 $A = 11, 3, 44, 22, 35, 6, 33,$

Să observăm că nu avem nevoie de un spațiu suplimentar pentru rezultat și putem face ceea ce se numește sortare “in place” (pe loc) (i.e. în afara șirului nu se țin decât cel mult un număr constant de elemente ale șirului) în pseudocod algoritmul poate fi scris astfel:

Start Sortins

1. citește(  $n, (A(i), i = 1, n)$  )
2. pentru  $i = 2, n$
3.  $k = A(i)$  {inserează  $A(i)$  în șirul sortat  $A(1), \dots, A(i - 1)$ }
4.  $j = i - 1$
5. cât timp  $j > 0$  și  $A(j) > k$
6.  $A(j + 1) = A(j)$
7.  $j = j - 1$
- sfârșit cât timp
8.  $A(j + 1) = k$  {a terminat inserarea}
- sfârșit 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 (instanță) este exprimat prin numărul de pași elementari executați.

Pentru început vom considera ca pas elementar fiecare linie parcursă din algoritmul în pseudocod și timpul pentru un astfel de pas o constantă diferită pentru fiecare linie (deși este evident că unele din aceste constante sunt egale) începem prin a prezenta algoritmul *Sortins* cu timpul fiecărei instrucțiuni și numărul de execuții ale liniei respective. Vom nota cu  $t_i$  numărul de execuții ale ciclului de la linia 5 (evident că el este diferit pentru fiecare valoare a lui  $i$ ).

Număr linie	Instrucțiune	Timp	Număr de operații
	<u>Start</u> Sortins		
1	<u>citește</u> ( $n, (A(i), i = 1, n)$ )	$c_1$	$n+1$
2	<u>pentru</u> $i = 2, n$	$c_2$	$n-1$
3	$k = A(i)$	$c_3$	$n-1$

4	$j = i - 1$	$c_4$	$n-1$
5	<u>cât timp</u> $j > 0$ și $A(j) > k$	$c_5$	$\sum_{i=2}^n t_i$
6	$A(j+1) = A(j)$	$c_6$	$\sum_{i=2}^n (t_i - 1)$
7	$j = j - 1$	$c_7$	$\sum_{i=2}^n (t_i - 1)$
	<u>sfârșit cât timp</u>		
8	$A(j+1) = k$	$c_8$	$n-1$
	<u>sfârșit pentru</u>		
9	<u>scrie</u> ( $A(i), i = 1, n$ )	$c_9$	$n$
	<u>Stop</u>		

$$T(n) = c_1(n+1) + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \sum_{i=2}^n t_i + (c_6 + c_7) \sum_{i=2}^n (t_i - 1) + (c_6 + c_7) \sum_{i=2}^n (t_i - 1) + c_8(n-1) + c_9n$$

Cum putem să-l aflăm pe  $t_i$ ? O să ne punem, de fapt, problema intervalului în care se “plimbă”, studiind cele două cazuri care corespund limitelor intervalului.

*Cel mai rapid caz* se obține când șirul este gata sortat și deci în linia 5 vom avea  $A(j) \leq k$  pentru orice  $j = 2, \dots, n$  deci în acest caz  $t_i = 1$ , iar  $T(n) = a \cdot n + b$ , unde  $a, b$  constante, deci  $T$  funcție liniară de  $n$ .

*Cel mai “rău” caz* se obține când șirul 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 dacă  $t_i = i-1$ , atunci  $\sum_{i=2}^n (i-1) = \frac{n(n-1)}{2}$  și  $\sum_{i=2}^n (i-2) = \frac{(n-2)(n-1)}{2}$ , deci  $T(n) = an^2 + bn + c$  este o funcție pătratică de  $n$ .

Să vedem acum dacă sortarea se poate face altfel, eventual mai repede. Pentru aceasta vom adopta o altă metodă, “divide et impera”. Această metodă pornește 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 șir format dintr-un singur element. Principiul “divide et impera” funcționează astfel:

1. se împarte problema în mai multe subprobleme
2. se rezolvă fiecare din subprobleme (în general recursiv)
3. se combină soluțiile pațiale într-o soluție finală

Vom reveni cu amănunte despre această metodă mai târziu.

Algoritmul *sortării prin interclasare* (mergesort) se bazează pe metoda “divide et impera”. Să vedem ce înseamnă interclasarea: din două șiruri sortate crescător să se creeze șirul sortat tot crescător care conține toate elementele celor două șiruri inițiale.

Aceasta dă următorul algoritm:

Start Interclasare

citește(  $n$ , ( $A(i)$ ,  $i = 1, n$ ) )

citește(  $m$ , ( $B(j)$ ,  $j = 1, m$ ) )

$k = 0$

$i = 1$

$j = 1$

repetă

$k = k + 1$

dacă  $A(i) < B(j)$  atunci

$C(k) = A(i)$

$i = i + 1$

altfel

$C(k) = B(j)$

$j = j + 1$

sfârșit dacă

până când  $i > n$  sau  $j > m$

{Ciclul de mai jos poate să nu se execute nici o dată}

pentru  $il = i, n$

$k = k + 1$

$C(k) = A(il)$

sfârșit pentru

{Ciclul de mai jos poate să nu se execute nici o dată}

pentru  $jl = j, m$

$k = k + 1$

$C(k) = B(jl)$

sfârșit 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 execuție pentru algoritmul de interclasare este  $T(n, m) = c(n + m)$  unde  $c$  este timpul maxim necesar execuției instrucțiunilor din interiorul ciclurilor *repetă* și *pentru* ( $c$  nu depinde de  $m, n$ ).



Urmăriți, într-un tabel, execuția algoritmului de mai sus. cu datele de intrare:

$A = 3, 5, 66, 77, 109, 304$ , iar  $B = 2, 4, 6, 8, 33, 55, 66, 88$

Pentru algoritmul de sortare prin interclasare prezentăm noțiunea de subalgoritm.



## Algoritmi parametrizați (subalgoritmi)

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

Pentru punerea în evidență a unei astfel de structuri modulare vom folosi algoritmi parametrizați numiți și subalgoritmi. Parametrii vor fi variabile ale căror valori vin din algoritmul care cheamă și se întorc înapoi în acesta, deci efectele produse asupra parametrilor de intrare vor exista și după ce subalgoritmul se termină.

Algoritmul *Interclas* parametrizat ar fi:  $\text{Interclas}(A, n, B, m, C)$  ceea ce înseamnă: “interclasează  $A$  (având  $n$  componente) cu  $B$  (având  $m$  componente) și obține  $C$ ”.

Într-un algoritm parametrizat nu mai au ce căuta instrucțiunile de intrare și nici instrucțiunea *Stop*; în locul acesteia din urmă vom folosi *Return* care înseamnă “întoarce-te în (sub)algoritmul chemător”.

Exemplu:  $\text{Interclas}(A, n, B, m, C)$ .

Correspondența dintre parametrii de chemare și cei din subalgoritm se face pozițional.

Pentru ușurarea exprimării, vom considera și subalgoritmi de tip funcție care returnează o valoare. În acest caz, valoarea care se determină de către subalgoritm este atribuită numelui funcției.



### Exemplu

$\text{ArieCerc}(r)$

$\text{ArieCerc} = \pi * r * r$

*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:

(Relativ la limbaje de programare: transmiterea parametrilor se face prin adresă.)

$\text{STotal} = \text{ArieCerc}(10) * 5$



Să ne reamintim...

**Algoritmii parametrizați (subalgoritmii) sunt algoritmi care sunt apelați din alt algoritm prin parametri care sunt și de intrare și de ieșire și se termină în return (revenire în algoritmul chemător).**

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

Așa cum am precizat la începutul acestui capitol algoritmul sortării prin interclasare *Mergesort* se bazează pe metoda “divide et impera”, ai cărei pași au fost deja prezentați. Evidențiem în continuare pașii 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

PAS1. împărţire: 2 1 5 3

PAS2. sortare (recursiv)

(a) împărţire: 2 1

(b) Sortarea este evidentă, fiecare parte având câte un element

(c) interclasare 1 2

(a) împărţire: 5 3

(b) Sortarea este evidentă, fiecare parte având câte un element

(c) interclasare 3 5

PAS3. interclasare: 1 2 3 5



Urmăriţi acest algoritmul pentru:

A= 3,5,4,8,11,12,10,2

Algoritmul va fi:

Mergesort( $A, p, r$ )

dacă  $p < r$  atunci

$q = \lfloor \frac{p+r}{2} \rfloor$

$2 \times \{ \text{partea întregă} \}$

$\{q \text{ este indicele elementului de la jumătatea şirului}\}$  *cheamă* Mergesort( $A, p, q$ )

*cheamă* Mergesort( $A, q + 1, r$ )

*cheamă* Merge( $A, p, q, r$ )

sfârşit dacă

Return

Acest algoritmul 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  $n/2$  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 dimensiuni care diferă printr-o unitate (subşirurile sunt cât mai echilibrate ca număr de elemente).



Verificați execuția algoritmului pentru șirul 3 2 5 6 4 1 0 7

### Notăția asimptotică

Pentru a vedea cum se comportă timpul de execuție al unui algoritm pentru intrări mari vom defini funcțiile  $\Theta$ ,  $\Omega$ ,  $O$ .

Spunem că  $T(n) = \Theta(f(n))$  dacă  $\exists N, c1, c2 > 0$  astfel încât:  $0 < c1f(n) < T(n) < c2f(n), \forall n > N$

Spunem că  $T(n) = \Omega(f(n))$  dacă  $\exists N, c1 > 0$  astfel încât:  $0 < c1f(n) < T(n), \forall n > N$  3.2)

Spunem că  $T(n) = O(f(n))$  dacă  $\exists N, c1 > 0$  astfel încât:  $T(n) < c1f(n), \forall n > N$

Lăsăm în seama cititorului să demonstreze proprietățile de mai jos.

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

#### 1. Tranzitivitate:

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

#### 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  $\exists M$  astfel încât  $g(n) > f(n), \forall n > M$  atunci  $\Theta(f(n) + g(n)) = \Theta(g(n))$

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

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

Aplicând proprietățile 1-7 se demonstrează imediat următoarele:

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

$$2. T(n) = c1 \cdot n \cdot \log n + c2 \cdot n + c3 \Rightarrow T(n) = \Theta(n \cdot \log n)$$

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

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

Când un algoritm conține o apelare a lui însuși se numește recursiv. În acest caz timpul de execuție se descrie printr-o ecuație recursivă. Pentru rezolvarea unei asemenea ecuații vom folosi metode matematice care vor fi descrise în capitolul dedicat metodei “Divide et impera”. Ne vom limita acum la a scrie recurența pentru sortare prin interclasare.

Dacă timpul pentru un șir cu  $n$  elemente este  $T(n)$ , atunci timpul pentru șirul cu  $n/2$

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

Mai departe 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 inserție ( $\Theta(n^2)$ ) este mai mare decât cea a algoritmului Mergesort ( $\Theta(n \log_2 n)$ ), altfel spus *Mergesort* este mai performant decât sortarea prin inserție.



### Rezumat

Pentru compararea a doi algoritmi, cel mai utilizat criteriu este cel al timpului de execuție relativ la dimensiunea datelor de intrare. Calculul complexității unui algoritm este o etapă obligatorie a soluției acestuia. Pentru exprimarea complexității se folosesc notațiile  $O(\cdot)$ ,  $\Omega(\cdot)$ ,  $\Theta(\cdot)$  care sunt legate de viteza de variație a timpului cerut pentru rezolvarea unei instanțe, în funcție de dimensiunea intrării și 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 interfețe (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.



### M1.U3.3. Test de autoevaluare a cunoștințelor

1. Determinați complexitatea algoritmilor din unitatea de învățare precedentă..
2. Se dau  $n$ ,  $((A(i, j), j = 1, n), i = 1, n)$  ( $A$  este o matrice pătratică, cu  $n$  linii și  $n$  coloane).

Să se calculeze produsul elementelor diferite de 0: (a) din toată matricea;  
(b) de pe diagonala principală;  
(c) de sub diagonala principală.

## ***M1.U4. Structuri elementare***



### **M1.U4.1. Introducere**

Aproape în orice algoritm se folosesc structuri suplimentare. Cele mai uzuale sunt prezentate aici.



### **M1.U4.2. Obiectivele unității de învățare**

. La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- folosească structurile liniare stiva și coada
- să folosească arborii binari.



**Durata medie de parcurgere a acestei unități de învățare este de trei ore.**

## **Structuri liniare**

O structură liniară este o mulțime de  $n \geq 0$  componente  $x(1), x(2), \dots, x(n)$  cu proprietățile:

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  $k \in \{2, \dots, n-1\}$  există un predecesor  $x(k-1)$  și un succesor  $x(k+1)$ .

Ne va interesa să executăm cu aceste structuri următoarele operații:

- 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 proprietăți;
- operații specifice

### **Stiva**

Una din cele mai cunoscute structuri liniare este stiva. O stivă este caracterizată prin *disciplina de intrare și de ieșire*. Să considerăm o mulțime de cărți puse una peste alta; există o primă carte care se poate lua foarte ușor (TOP) și o carte care se poate lua numai dacă se înlătură toate celelalte cărți (BOTTOM).

Disciplina unei stive este “ultimul intrat, primul ieșit” (disciplina care nu v-ar place să fie respectată când stați la rând la lapte), prescurtat LIFO (Last In, First Out).

Diferențele față de un vector sunt:

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

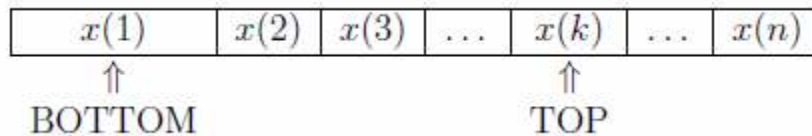
- o stivă poate fi vidă;
- stiva are un număr variabil de elemente în timpul execuției unui algoritm

Se pune problema reprezentării concrete a unei stive în memoria unui calculator.

Putem alocă o stivă în două moduri: secvențial sau înlănțuit.

### Alocarea secvențială 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:

$a \Rightarrow$  Stiva

dacă  $k = n$  atunci “Depășire”

altfel

$k = k + 1$

$x(k) = a$

sfârșit dacă

Return

Algoritmul de ieșire (scoatere) din stivă este:

$a \Leftarrow$  Stiva

dacă  $k = 0$  atunci “Stiva vidă”

altfel

$a = x(k)$

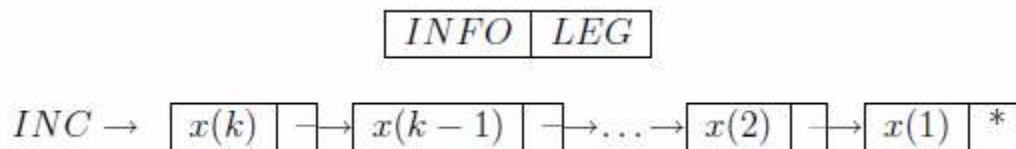
$k = k - 1$

sfârșit dacă

Return

### Alocarea înlănțuită a stivei

În alocarea înlănțuită fiecare element al structurii este însoțit 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 evidență (ancorat) în *INC* și elementul de la baza stivei va avea conține în câmpul *LEG*  $*$ , ca în figură:



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

În acest caz intrarea în stivă va folosi stiva de locuri libere1 (această stivă se numește *LIBERE*), pentru a obține noi locuri la introducerea în stivă. Vom prezenta în continuare algoritmi de intrare/ieșire dintr-o stivă în cazul în care alocarea ei a fost înălțuită. Algoritmul de intrare în stivă este:

$a \Rightarrow$  Stiva  $\{a$  este informația introdusă în stivă $\}$

$y \leftarrow LIBERE\{alocare\ de\ memorie\ pentru\ elementul\ nou,\}$

$INFO(y) = a$

$LEG(y) = INC$

$INC = y$

Return

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

Algoritmul de ieșire din stivă este:

$a \leftarrow Stiva\ \{a\}$  este informația extrasă din vârful stivei nevide $\}$

dacă  $INC = *$  atunci “Stiva este vidă”

altfel

$a = INFO(INC)$

$aux = INC$

$INC = LEG(INC)$

$aux \Rightarrow LIBERE\{Eliberarea\ spațiului\ de\ memorie\}$

sfârșit dacă

Return



Să ne reamintim...

**Stiva este o structură liniară cu disciplina ultimul intrat .. primul ieșit.**

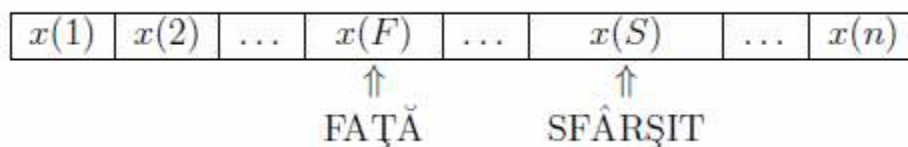
### Coadă

O altă structură de date folosită în concepția algoritmilor este coada. O coadă este caracterizată și ea de o disciplină de intrare/ieșire, bineînțeles diferită de cea a stivei. De data aceasta puteți să vă gândiți la coada la lapte: primul care s-a așezat 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). Cunoscută în cadrul limbajelor de programare drept *heap*; obținerea de memorie se face prin *alocare dinamică*, eliberarea ei prin *dealocare*. O coadă poate fi vidă și are și ea un număr variabil de elemente în timpul execuției unui algoritm.

### Alocarea secvențială a cozii

Coadă alocată secvențial își va găsi locul tot într-un vector ( $X(i)$ ,  $i = 1, n$ ):



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

Algoritmul de intrare în coadă este:

$a \Rightarrow Coadă$

dacă  $S = n$  atunci “Depășire”

altfel

$S = S + 1$

$x(S) = a$

sfârșit dacă

Return

Algoritmul de ieșire din coadă este:

$a \Leftarrow Coadă$

dacă  $F > S$  atunci “Coadă este vidă”

altfel

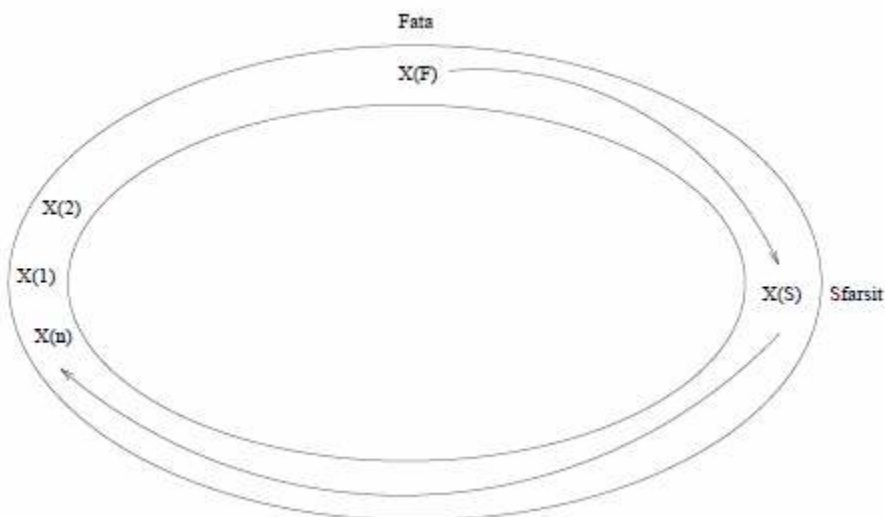
$a = x(F)$

$F = F + 1$

sfârșit dacă

Return

Se poate imagina ușor că procedând în acest mod (scoțând din față și introducând la sfârșit), coada “migrează” spre dreapta și poate să ajungă în situația de depășire când de fapt mai există mult loc gol (în vectorul  $x$ ) pe primele poziții. Apare astfel ideea de a folosi elementele vectorului ca și cum ar fi dispuse circular, precum în figură

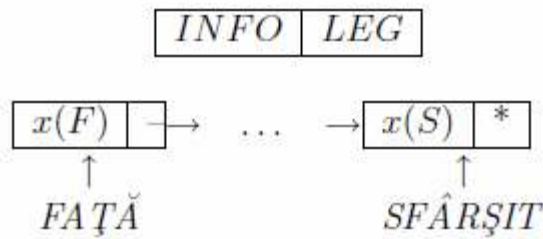




Lăsăm în seama cititorului să scrie algoritmi de intrare/ieșire din coada circulară.

### Alocarea înlănțuită a cozii

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



Algoritmul de intrare în coadă este:

$a \Rightarrow \text{Coadă}$

$y \leftarrow \text{LIBERE}$

$\text{INFO}(y) = a$

$\text{LEG}(y) = *$

dacă  $\text{FAȚĂ} = *$  atunci {Coadă este vidă}  $\text{FAȚĂ} = y$   
altfel

$\text{LEG}(\text{SFÂRȘIT}) = y$

sfârșit dacă

$\text{SFÂRȘIT} = y$

Return

Inițial, coada este vidă, i.e.  $\text{FAȚĂ} = \text{SFÂRȘIT} = *$ .

Algoritmul de ieșire din coadă este:

$a \Rightarrow \text{Coadă}$

dacă  $\text{FAȚĂ} = *$  atunci "Coadă este vidă"  
altfel

$a = \text{INFO}(\text{FAȚĂ})$

$\text{aux} = \text{FAȚĂ}$

$\text{FAȚĂ} \Rightarrow \text{LEG}(\text{FAȚĂ})$

$\text{aux} \Rightarrow \text{LIBERE}$

sfârșit dacă

Return



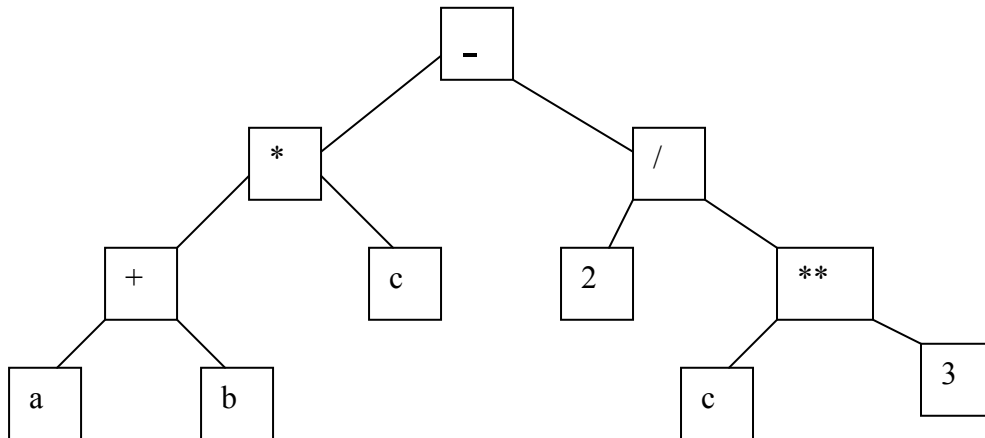
Să ne reamintim...

**Coadă este o structură liniară cu disciplina primul intrat primul ieșit.**

## Arbori binari

Un arbore în care orice vârf (nod) are 0, 1 sau 2 descendenți se numește *arbore binar*. Un arbore binar care are vârfuri cu 0 sau 2 descendenți se numește *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:



Unde \*\* este semnul pentru ridicarea la putere.

Frunzele acestui arbore (nodurile fără descendenți) conțin operanzii, iar celelalte noduri conțin operatorii. Deoarece nu toate operațiile 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:

- secvențial
- înlănțuit
- mixt

### Alocarea secvențială 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 din stânga este  $x(2i)$  iar cel din dreapta este  $x(2i + 1)$ ;
- dacă nu există descendent, se pune \*.

Reprezentarea secvențială 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

*Observație:* Pozițiile  $x(10), \dots, x(13), x(16), \dots, x(31)$  conțin \*.

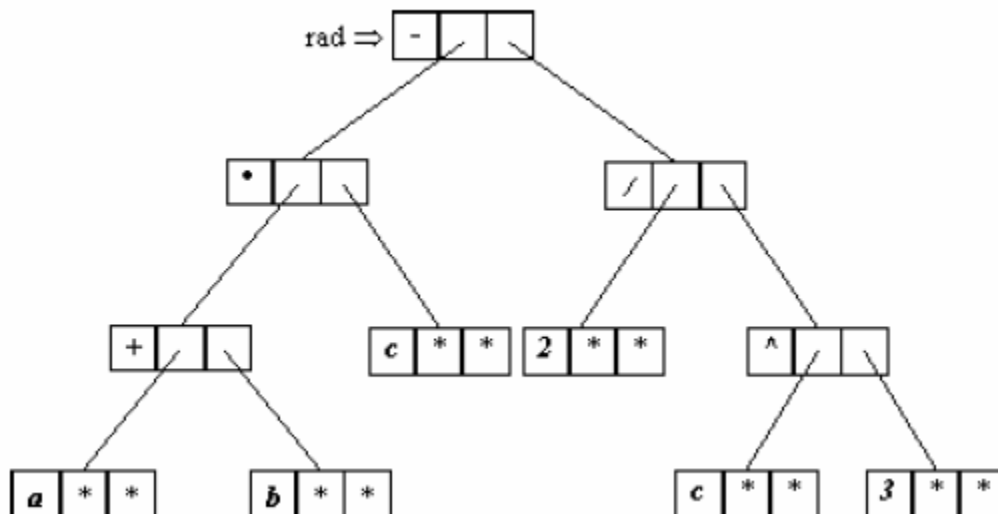
Vom folosi această alocare în capitolul destinat sortării pentru a construi o structură de date asemănătoare, numită HEAP.

### Alocarea înlănțuită a unui arbore binar

În acest caz se folosesc noduri de forma 

INFO	LS	LD
------	----	----

 unde *LS* și *LD* conțin adresele de memorie ale nodurilor descendent stâng, respectiv descendent drept. Reprezentarea înlănțuită a arborelui din figura de mai sus este dată în aici



### Alocarea mixtă a unui arbore binar

La acest tip de alocare se folosesc trei vectori *INFO*, *LS*, *LD* unde *LS(i)* și *LD(i)* conțin indicii unde se află memorări descendentul stâng, respectiv descendentul drept al nodului *i*. Reprezentarea mixtă a acestui arbore este:  
*INFO LS LD*

	<i>INFO</i>	<i>LS</i>	<i>LD</i>
1	-	2	3
2	.	4	5
3	/	6	7
4	+	8	9
5	c	*	*
6	2	*	*
7	^	10	11
8	a	*	*
9	b	*	*
10	c	*	*
11	3	*	*

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

- arborele binar se definește recursiv ca: rădăcină, subarbore stâng, subarbore drept (fiecare subarbore este la rândul lui un arbore binar, posibil vid);
- mai întâi se va parcurge subarborul stâng apoi cel drept.

După poziția rădăcinii ( $R$ ) relativ la subarborele stâng ( $S$ ) și 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*:

- +  $a$   $b$   $c$  / 2 \*\*  $c$  3 această scriere se numește *scriere poloneză prefixată* a unei expresii aritmetice.

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

$a + b$   $c - 2 / c$  \*\* 3

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

$a$   $b + c$  2  $c$  3 \*\* / - această scriere se numește *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 conține nodurile succesive parcurse pe stânga care urmează să fie parcurse și pe dreapta; rădăcina arborelui se află în *rad*.

Preordine iterativ(*rad*)

$STIVA = \emptyset$  {STIVA este vidă}  $nodCurent = rad$

$maiAmNoduri = adevarat$

repetă

$parcurePeStanga = adevarat$

repetă

dacă  $nodCurent \neq \text{nil}$  atunci

scrie( INFO( $nodCurent$ ) )

{parcure pe stânga}

$nodCurent \Rightarrow STIVA$

$nodCurent = LS(nodcurent)$

altfel

$parcurePeStanga = fals$

sfârșit dacă

până când  $parcurePeStanga = fals$

{parcure pe dreapta}

dacă  $STIVA = \emptyset$  atunci

$maiAmNoduri = fals$

altfel

$nodCurent \Leftarrow STIVA$

$nodCurent = LD(nodCurent)$

sfârșit dacă

până când  $maiAmNoduri = adevarat$

Return

Se pot da algoritmi iterativi și pentru celelalte două parcurgeri (vezi problema 2 de la exerciții). De asemenea se pot da algoritmi recursivi pentru parcurgeri, mai ușor de înțeles, dar ceva mai ineficienți.



### 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 și coada au politici specifice de acces, iar arborele poate fi parcurs în moduri diferite pentru procesarea datelor.



### M1.U4.3. Test de autoevaluare a cunoștințelor

1. Scrieți algoritmi de intrare/ieșire din coada circulară.
2. Scrieți algoritmi iterativi pentru parcurgerea arborilor binari în inordine și postordine.
3. Se dă o stivă alocată înlăntuit și o valoare  $x$ . Să se extragă din stivă, dacă există, elementul cu  $INFO = x$ .
4. Se dă o stivă și 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.
7. Să se dea algoritmi recursivi pentru parcurgerea în preordine, postordine, inordine a unui arbore binar.

## ***M1.U5.Recursivitate..***



### **MI.U5.1. Introducere**

Recursivitatea reprezintă un mod de elaborare a algoritmilor, caracterizat prin faptul că un subalgoritm se apelează pe el însuși (direct sau indirect).



### **MI.U5.2. Obiectivele unității de învățare**

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- înțeleagă modul de lucru cu algoritmi recursivi
- construiască corect algoritmi recursivi



**Durata medie de parcurgere a acestei unități de învățare este de trei ore.**

Recursivitatea a apărut din necesități practice, permițând implementarea unei funcții matematice recursive. Nu reprezintă o modalitate mai puternică sau mai slabă din punct de vedere al rezultatelor ce se pot obține comparativ cu metodele iterative; s-a demonstrat că aceste două abordări sunt egale ca și putere de calcul. Algoritmii recursivi nu sunt mai rapizi decât algoritmii iterativi echivalenți, dar uneori sunt mai expresivi și mai ușor de întreținut (modificat). Pentru înțelegerea algoritmilor recursivi, este esențială înțelegerea lucrului cu stivă. Înainte de a se efectua un apel de subalgoritm (recursiv sau nu), se salvează pe stivă adresa de memorie a instrucțiunii de la care se va continua execuția după întoarcerea din apel. La intrarea într-un subalgoritm se rezervă spațiu pe stivă pentru parametrii transmiși și pentru variabilele care au fost declarate în subalgoritm (variabile locale). La ieșirea din apel, stivă se eliberează de informația depusă la intrarea în apel, iar adresa următoarei instrucțiuni ce se va executa se preia de pe vârful stivei (și de asemenea această informație va fi scoasă de pe stivă).

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) și nu prin adresă (când pe stivă se copiază adresa de memorie a parametrilor efectivi). În cazul exemplului de mai jos, întrucât în corpul subalgoritmului nu se modifică valoarea lui  $n$ , rezultatul ar fi același indiferent de convenția de apel.



Start Exemplu

*citește( n )*

Recursiv(*n*)

*scrie( "Sfârșit" )*

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 )

sfârșit dacă

Return



Verificați execuția pentru *n* egal cu 3..

Ca și comentarii sunt trecute niște adrese la care se presupune a se afla instrucțiunile corespunzătoare. Să presupunem că se va citi  $n = 2$ . Evoluția stivei programului este:

- La intrarea în algoritmul *Exemplu*, se rezervă spațiu 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 instrucțiunii care se va executa după ce se va reveni din acest apel (în cazul nostru instrucțiunea de scriere cu adresa 10):

*adr* = 10

*n* = 2

- Se apelează *Recursiv*(2); se scrie pe stivă valoarea parametrului  $n = 2$ , se rezervă spațiu 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 condiția nu este îndeplinită, se intră pe ramura *altfel*.

- Avem din nou apel: înainte de acesta se salvează adresa de memorie a instrucțiunii de la care se va continua execuția (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 spațiu 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 condiția nu este îndeplinită, se intră pe ramura *altfel*. •

Avem din nou apel: înainte de acesta se salvează adresa de memorie a instrucțiunii de la care se va continua execuția (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 spațiu 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 condiția îndeplinită, se intră pe ramura *atunci*, care reprezintă întoarcere la metoda *Recursive(1)*. • La întoarcerea din *Recursive(0)* în *Recursive(1)*, se scoate de pe stivă informația care a fost depusă datorită apelului *Recursive(0)*, revenindu-se la forma de dinainte de efectuarea acestui apel:

$i = 2$

$n = 1$

$adr = 100$

$i = 4$

$n = 2$

$adr = 10$

$n = 2$

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

• Suntem în *Recursive(1)*. Execuția se continuă de la instrucțiunea cu adresa 100 (conform punctului precedent), deci  $i$  ia valoarea  $2+1$  (valoarea 2 din sumă fiind datorată informației  $i = 2$  de pe stivă), care se afișează.

• Datorită instrucțiunii de retur finale din subalgoritmul *Recursive*, avem o întoarcere de la *Recursive(1)* la *Recursive(2)*. Se curăță stiva de informațiile  $i = 2$ ,  $n = 1$ ,  $adr = 100$  și se continuă execuția de la instrucțiunea cu adresa 100:

$i = 4$

$n = 2$

$adr = 10$

$n = 2$

• Suntem în *Recursive(2)*. Execuția se continuă de la instrucțiunea cu adresa 100, deci  $i$  ia valoarea  $4+1$  (valoarea 4 din sumă fiind datorată informației  $i = 4$  de pe stivă), care se afișează.

• Datorită instrucțiunii de retur finale din subalgoritmul *Recursive*, avem o întoarcere de la *Recursive(2)* la *Exemplu*. Se curăță stiva de informațiile  $i = 4$ ,  $n = 2$ ,  $adr = 10$  și se continuă execuția de la instrucțiunea cu adresa 10. Stiva va fi:

$n = 2$

• Se execută instrucțiunea cu adresa 10: *Scrie("Sfarșit")*.

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

Algoritmul a afișat valorile 3, 5. Evident, acest lucru se putea face mult mai simplu printr-o iterație; dar exemplul de față a dorit să ilustreze funcționarea unui apel recursiv.

Datorită operațiilor de curățare a stivei (care se efectuează printr-o simplă incrementare sau decrementare a valorii dintr-un registru al microprocesorului), precum și datorită informațiilor repetate care se salvează pe stivă ( $n, i, adr$ ), timpul și spațiul de memorie necesar sunt mai mari decât pentru variantele iterative. Totuși, în destul de multe cazuri implementările recursive sunt mult mai ușor de urmărit și de înțeles decât cele iterative.

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

Dacă revenim la convenția ca transferul parametrilor să se facă prin adresă, atunci pe stivă, în cadrul 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).



### Exemplu

#### Calcularea lui $n!$

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

$n! = 1$ , pentru  $n = 0$

și  $n! = n \cdot (n - 1)!$ , pentru  $n > 0$

Algoritmul recursiv este următorul:

*Start* FactRec

*citește*(  $n$  )

*scrie*(  $Fact(n)$  )

*Stop*

unde subalgoritmul *Fact* are forma:

$Fact(n)$

*dacă*  $n = 0$  *atunci*

$Fact = 1$

*altfel*

$Fact = n * Fact(n - 1)$

*sfârșit dacă*

*Return*



Scrieți algoritmul de mai sus nerecursiv

Complexitatea algoritmului este dată de ecuația recursivă  $T(n) = T(n - 1) + c$ ,  $c$  fiind o constantă datorată apelului recursiv (operației cu stiva) și înmulțirii, pentru care soluția este  $T(n) = \Theta(n)$ . Varianta iterativă este de aceeași complexitate teoretică, dar mai rapidă în practică.



### Exemplu

#### Șirul lui Fibonacci

Șirul lui Fibonacci se definește astfel:

$$f(n) = 0, \text{ pentru } n = 0$$

$$f(n) = 1, \text{ pentru } n = 1$$

$$f(n) = f(n - 1) + f(n - 2), \text{ pentru } n \geq 2$$

Vom implementa recursiv acest algoritm:

*Start FiboRec*

*citește( 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)*

*sfârșit dacă*

*Return*



Complexitatea teoretică este dată de ecuația recursivă

$$T(n) = T(n - 1) + T(n - 2) + c,$$

care duce la  $T(n) = \Theta(\varphi^n)$ , unde  $\varphi = 1 + \sqrt{5} / 2$  (tăietura de aur). Complexitatea exponențială mare este datorată faptului că se calculează de mai multe ori aceleași 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 și pentru cel de al doilea membru al părții drepte a ecuației anterioare. Redundanța de calcul devine tot mai accentuată pentru valori mici ale argumentului.

Calculați.



### Exemplu

Să vedem o implementare iterativă:

*Start FiboIter*

*citește( n )*

$$f(0) = 0$$

$$f(1) = 1$$

*pentru i = 2, n*

$$f(i) = f(i - 1) + f(i - 2)$$

*sfârșit pentru  
scrie( $f(n)$ )  
Stop*



Aplicați algoritmul pentru șirul  $n=5$

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



### Rezumat

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



### M1.U5.3. Test de autoevaluare a cunoștințelor

1. Să se calculeze recursiv maximul unui șir.
2. Să se calculeze recursiv suma elementelor dintr-un șir.
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 înmulțiri.
5. Să se scrie un algoritm care calculează termenul  $fibonacci(n)$  al șirului Fibonacci în timp mai bun de  $\Theta(n)$  (a se vedea și problema 3).
6. Să se dea algoritmi recursivi pentru parcurgerea în preordine, postordine, înordine a unui arbore binar.

## Modulul 2. Metode de elaborare a algoritmilor.

---

### Cuprins

Introducere .....	36
U1. Metoda Backtracking. ....	36
U2. Generarea submulțimilor. ....	41
U3. Metoda Divide et Impera .....	53
U4. Sortare și statistici de ordine .....	61
U5. Metoda Greedy .....	71
U6. Metoda programării dinamice .....	81



### Introducere.

În timp s-au concretizat metode generale pentru elaborarea algoritmilor. Modulul acesta introduce cele mai folosite dintre aceste metode.

### M2.U1. Metoda Backtracking.



#### M2.U1.1. Introducere

Acest capitol introduce una din cele mai generale modalități de rezolvare a problemelor: metoda Backtracking. Ea duce la scrierea unor algoritmi care evită generarea tuturor combinațiilor posibile și apoi la validarea rezultatelor.

Este prezentată schema generală de algoritm backtracking și probleme clasice pentru a căror rezolvare se folosește această abordare.



#### M2.U1.2. Obiectivele unității de învățare

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- urmărească parcurgerea incompletă a unui arbore de generare de soluții
- să construiască algoritmi iterativi prin metoda Backtracking.



**Durata medie de parcurgere a acestei unități de învățare este de două ore.**

Metoda Backtracking se poate aplica algoritmilor care trebuie să construiască o soluție de forma  $\mathbf{x} = (x(1), x(2), \dots, x(n))$  unde pentru orice  $i \in \{1, 2, \dots, n\}$ ,  $x(i) \in A_i$ ,  $A_i$  fiind o mulțime dată ca informație de intrare și unde, în plus, componentele lui  $\mathbf{x}$  trebuie să mai

satisfacă niște condiții interne.



### Exemple

Să considerăm:  $A1 = \{A, B, E\}$ ,  $A2 = \{A, C\}$ ,  $A3 = \{U, S, E, C\}$

Cerințe:

1. Să se construiască cuvintele care au prima literă din  $A1$ , a doua literă din  $A2$ , a treia literă din  $A3$ ; și care respectă condițiile interne de la 2;
2. în plus, se cere respectarea următoarelor condiții interne:
  - (a) Nu sunt două litere consecutive egale;
  - (b) Nu sunt formate numai din consoane sau numai din vocale.

Dacă luăm în considerare numai cerința de la 1, obținem *soluțiile admisibile* ale problemei, de exemplu (AAU), (ACS), (EAU), (BAU), (BCC), . . . .

Dar (AAU), (BCC) nu respectă condițiile interne 2a, iar (EAU) nu respectă condiția 2b.

O metodă de rezolvare a acestei probleme ar fi să construim mai întâi toate soluțiile admisibile și apoi să verificăm care dintre ele respectă condițiile interne. Acest mod de a aborda problema duce, în exemplul nostru, la  $3 \cdot 2 \cdot 4 = 24$  de soluții din care evident cele care încep cu AA, de exemplu, nu sunt bune. Dacă mulțimile  $A1, A2, \dots, An$  au respectiv  $k1, k2, \dots, kn$  elemente, atunci am avea în total  $k1 \cdot k2 \cdot \dots \cdot kn$  soluții admisibile. Pentru  $k1 = k2 = \dots = kn = m$  avem  $m^n$  soluții admisibile, deci un algoritm de tip *brute force* care ia în considerare toate posibilitățile ar avea o complexitate exponențială.



Studiați aceeași problemă cu mulțimile:  $A1 = \{A, C, E\}$ ,  $A2 = \{A, C\}$ ,  $A3 = \{A, S, E, C\}$

Algoritmul de tip backtracking urmărește micșorarea numărului de operații, dar trebuie să spunem că rămâne de complexitate exponențială. Acest algoritm urmărește să genereze câte o componentă  $x(k) \in Ak$  cât timp mai are o componentă netestată din  $Ak$ ; dacă nu mai are componente netestate, se revine la alegerea elementului de pe poziția anterioară  $x(k-1)$ , altfel se merge la componenta  $x(k+1)$ . În felul acesta s-ar genera toată mulțimea soluțiilor admisibile; dar dacă după ce am găsit o componentă  $x(k)$  testăm dacă bucata de vector  $(x(1), x(2), \dots, x(k))$  generată până atunci satisface *condițiile de continuare* derivate din condițiile interne, atunci vom putea să nu permitem generarea inutilă a unor soluții care oricum nu ar fi bune (în exemplul nostru (AA)).

Trebuie să mai subliniem că simplul fapt că bucata  $(x(1), x(2), \dots, x(k))$  din vectorul  $x$  satisface condițiile de continuare nu înseamnă că o să găsim, obligatoriu, o soluție a problemei (există, totuși, o metodă care găsește soluția 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 soluție}

$k = k - 1$

altfel

dacă mai există valori netestate  $a \in A_k$  atunci

$x(k) = a$

dacă  $(x(1), x(2), \dots, x(k))$  satisface condițiile de continuare

atunci

$k = k + 1$

sfârșit dacă

altfel

$k = k - 1$

sfârșit dacă

sfârșit dacă

sfârșit cât timp

Return



### Exemplu

Vom utiliza acest algoritm prin problema așezării damelor pe tabla de șah astfel încât oricare două dame să nu se atace (2 dame se atacă dacă sunt pe aceeași linie, coloană, diagonală). Vom considera mulțimile  $(A_i)_{i=1,n}$  ca fiind pozițiile (de la 1 la  $n$ ) pe care le poate ocupa o damă pe linia  $i$ . Evident că de data aceasta toate mulțimile  $A_i$  sunt egale. Deci o configurație pe o tablă de  $n = 5$  precum mai jos va fi simbolizată de vectorul soluție  $\mathbf{x} = (1, 3, 5, 2, 4)$  și se poate vedea că în acest caz condițiile interne sunt respectate (oricare două dame nu se bat).

😊				
		😊		
				😊
	😊			
			😊	

Algoritmul pentru  $n > 4$  este:

Dame( $n$ )

pentru  $i = 1, n$

$x(i) = 0$  {inițial pe nici o linie nu avem poziționată nici o regină}

sfârșit 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  $x(k) \leq n$  atunci  
             dacă  $\text{DispunereCorecta}(x, k) = \text{adevarat}$   
                 atunci  
                      $k = k + 1$   
                 sfârșit dacă  
                     altfel  
              $x(k) = 0$  { Nu avem nici o regină pe  
             această linie }  $k = k - 1$  { Ne întoarcem la  
             regina de pe poziția anterioară } sfârșit  
             dacă {  $x(k) \leq n$  }  
         sfârșit dacă {  $k = n + 1$  }  
     sfârșit cât timp

#### Return

DispunereCorecta( $x, k$ ) testează dacă condițiile de continuare pentru bucata  $x(1), \dots, x(k)$  sunt îndeplinite; returnază adevărat dacă nu avem regine care se atacă, fals în caz contrar.

DispunereCorecta( $x, k$ )

*corect = adevarat*

{ presupunem că nu avem nici un atac între regina  $k$  și cele dispuse anterior }

pentru  $i = 1, k - 1$

dacă  $x(k) = x(i)$  sau  $|x(k) - x(i)| = k - i$  atunci

        { sunt pe aceeași coloană sau pe aceeași diagonală }

*corect = fals*

sfârșit dacă

sfârșit pentru

*DispunereCorecta = corect*

#### Return

Aplicați algoritmul de mai sus pentru  $n=6$ .



#### **Rezumat**

Multe probleme cer generarea uneia, mai multora sau tuturor soluțiilor care respectă anumită cerințe. O abordare de tipul “generează toate combinațiile, apoi reține-le pe cele valide” (*brute-force*) este cea mai ineficientă. Metoda backtracking urmărește depistarea cât mai devreme cu putință a soluțiilor care sunt invalide (pe măsură ce componentele lor se generează). Deși algoritmi



rezultați sunt de complexitate exponențială, multe soluții invalide sunt evitate.



### **M2.U1.3. Test de autoevaluare a cunoștințelor**

1. Scrieți un algoritm pentru parcurgerea tablei de șah cu un cal (calul sare pe diagonală unui dreptunghi cu laturile de un pătrat și două patrate). Fiecare pătrat al tablei trebuie să fie vizitat exact o singură dată.
2. Având 4 culori și o hartă cu  $n$  țări (dată print-o matrice de adiacență:  $a_{ij} = 1$  dacă țara  $i$  este vecină cu țara  $j$ , 0 altfel), să se coloreze harta astfel ca două țări vecine să nu aibe aceeași culoare (două țări se consideră a fi vecine dacă au o frontieră comună).
3. O organizație are în componența sa  $n$  bărbați și  $m$  femei. Să se scrie un algoritm care listează toate modalitățile în care se poate alcătui o delegație care să conțină cel puțin  $k$  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  $b(i)$  bucăți? Să se dea toate soluțiile posibile.

## M2.U2. Generarea submulțimilor



### M2.U2.1. Introducere

De multe ori soluțiile care se cer pentru rezolvarea unei (sub)probleme sunt de o formă particulară: produse carteziane, familia submulțimilor, a combinațiilor, aranjamentelor, permutărilor. Acest capitol conține rezolvări clasice pentru aceste probleme, în mai multe variante



### M2.U2.2. Obiectivele unității de învățare

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- detecteze o problemă de generare
- formuleze modelul informatic al problemei
- scrie algoritmul recursiv sau iterativ de rezolvare al problemei



**Durata medie de parcurgere a acestei unități de învățare este de trei ore.**

### Schema generală de lucru

O aplicație imediată a metodei Backtracking este generarea submulțimilor.

Vom considera submulțimi ale mulțimii  $A = \{a(1), a(2), \dots, a(n)\}$  cu  $n$  elemente.

Evident că avem bijecție între mulțimea indicilor  $\{1, 2, \dots, n\}$  și elementele mulțimii  $A$ .



### Exemple

$\{1, 3, 4, 5\} \leftrightarrow \{a(1), a(3), a(4), a(5)\}$  Ca atare, vom presupune pentru simplificarea algoritmilor că  $a(i) = i, \forall i \in \{1, \dots, n\}$ .

Avem la dispoziție următoarele modalități de reprezentare a unei submulțimi  $X$  cu  $i$  elemente ale lui  $A$ :

1. Printr-un vector  $x$  cu  $i$  componente care conține elementele submulțimii, precizându-se valoarea lui  $i$  sau completând componentele lui  $x$  cu o valoare neutilizată (de exemplu 0);
2. Analog cu metoda 1, dar plasând elementele submulțimii la sfârșitul vectorului  $x$ ;
3. Prin vectorul caracteristic al submulțimii: un vector  $c$  cu  $n$  componente unde  $c(i) = 1$  arată că  $a(i)$  aparține submulțimii, iar  $c(i) = 0$  arată că  $a(i)$  nu aparține submulțimii.

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

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

2.  $x = (0, 0, 0, 1, 3, 4, 5)$ : elementele din submulțimea  $X$  la sfârșit, primele poziții fiind completate cu o valoare care nu se găsește în  $A$ ;
3.  $c = (1, 0, 1, 1, 1, 0, 0)$ :  $c$  este vectorul caracteristic:  $c(i) = 1$ , dacă  $i \in X$   $c(i) = 0$ , dacă  $i \notin X$

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

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

Generare( $x, n, ig$ )

dacă  $ig = 0$  atunci

$x$  = valoarea (submulțimea) inițială

$ig = 1$

Return {iese în (sub)algoritmul apelant}

sfârșit 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 submulțimea  $X$ }

sfârșit 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 {leși din ciclare}

altfel

cheamă Prelucrare( $x, n$ ) {prelucrează elementul proaspăt obținut al submulțimii}

sfârșit dacă

până când fals

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



Să ne reamintim...

Mulțimile sunt generate element cu element, după generare acesta se prelucrează. Folosim un indicator de generare are inițial valoarea 0; apoi are valoarea 1 cât timp se mai poate genera o submulțime  $X$  și din nou valoarea 0 când s-a terminat generarea.

### Generarea elementelor unui produs cartezian

Fiind date mulțimile  $A_1, A_2, \dots, A_m$ , unde  $A_i = \{1, 2, \dots, n(i)\}$ , dorim să generăm elementele produsului cartezian  $A_1 \times A_2 \times \dots \times A_m$ . 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 configurație  $x = (x(1), x(2), \dots, x(m))$  va desemna elementul  $(A_1(x(1)), A_2(x(2)), \dots, A_m(x(m)))$  al produsului cartezian.

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

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

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

Algoritmul pentru generarea produsului cartezian este:

ProdusCartezian( $x, m, n, ig$ )

dacă  $ig = 0$  atunci

pentru  $i = 1, m, 1$

$x(i) = 1$

sfârșit pentru

$ig = 1$

Return

sfârșit dacă

pentru  $i = m, 1, -1$

dacă  $x(i) < n(i)$  atunci

$x(i) = x(i) + 1$

Return

altfel

$x(i) = 1$

sfârșit dacă  
sfârșit pentru  
 $ig = 0$   
Return

### Generarea tuturor submulțimilor unei mulțimi

O primă metodă ar fi să generăm elementele produsului cartezian al mulțimilor  $A_1=A_2=\dots=A_n=\{0, 1\}$ . Vectorul  $x$  care se obține, interpretat ca vector caracteristic, va fi element al mulțimii părților lui  $\{1, 2, \dots, n\}$

O a doua metodă ve genera submulțimile unei mulțimi în ordinea crescătoare a numărului de elemente utilizând metoda (2) de reprezentare a mulțimilor. Specificăm că vectorii  $x$  vor fi generați și de această dată în ordine lexicografică strict crescătoare.

Fie  $x = (x(1), x(2), \dots, x(n))$ ; atunci succesorul lui se va determina în modul următor: se va determina indicele  $i$  cu proprietatea că  $x(i) < i$  și 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:

$$x'(i+1) = x(i) + 2, \dots, x'(n) = x(i) + n - i + 1.$$

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

Algoritmul pentru generarea submulțimilor unei mulțimi este:

Submultimi( $x, m, ig$ )

dacă  $ig = 0$  atunci

pentru  $i = 1, n$   
 $x(i) = 0$

sfârșit pentru

$ig = 1$

Return

sfârșit dacă

pentru  $i = n, 1, -1$

dacă  $x(i) < i$  atunci

$$x(i) = x(i) + 1$$

pentru  $j = i + 1, n$

$$x(j) = x(j - 1) + 1$$

sfârșit pentru

Return

sfârșit dacă  $\{x(i) < i\}$

sfârșit pentru

$ig = 0$

Return

### Generarea mulțimii combinărilor

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

O primă metodă se bazează pe metoda de reprezentare 1. Și de această dată vectorii vor fi generați în ordine lexicografică începând cu  $x = (1, 2, \dots, m)$ .

Fiind dat  $x = (x(1), x(2), \dots, x(m))$ , următorul element  $x'$  va fi determinat astfel:

1. se determină indicele  $i$  cu proprietatea:  $x(i) < n - m + i, x(i+1) = n - m + i + 1, \dots, x(m) = n$
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  $x = (n - m + 1, n - m + 2, \dots, n)$  și deci s-a generat ultimul element.

Algoritmul va fi următorul:

Combin1( $x, n, m, ig$ )

dacă  $ig = 0$  atunci

pentru  $i = 1, m$

$x(i) = i$

sfârșit pentru

$ig = 1$

Return

sfârșit dacă

pentru  $i = m, 1, -1$

dacă  $x(i) < n - m + i$  atunci

$x(i) = x(i) + 1$

pentru  $j = i + 1, m$

$x(j) = x(j - 1) + 1$

sfârșit pentru

Return

sfârșit dacă

sfârșit pentru

$ig = 0$

Return

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

Fie acum vectorul  $x$  căruia vrem să îi determinăm succesorul  $x'$ ; pentru aceasta vom pune în

evidență ultima secvență de cifre 1:  $x = (\dots, \underset{a}{0}, \underset{b}{1}, \dots, 1, 0, \dots, 0)$  unde  $a$  este indicele

unde începe această secvență iar  $b$  indicele ultimului element egal cu 1.

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

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

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

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

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

$$x = (\dots, \underset{a}{1}, \underset{n-l_1+2}{0}, \dots, \underset{l_1-1}{1}, \underset{l_1-1}{0}, \dots, 0)$$

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

Combinari2( $x, n, m, ig, a, b$ )

dacă  $ig = 0$  atunci

pentru  $i = 1, n - m$

$x(i) = 0$

sfârșit pentru

pentru  $i = n - m + 1, n$

$x(i) = 1$

sfârșit pentru

$ig = 1$

$a = n - m + 1$

$b = n$

Return

sfârșit dacă

dacă  $a = 1$  atunci

$ig = 0$

Return

sfârșit dacă

dacă  $a < b$  atunci

$l_1 = b - a + 1$

$x(a - 1) = 1$

pentru  $i = a, n - l_1 + 1$

$x(i) = 0$

sfârșit pentru

pentru  $i = n - l_1 + 2, n$

```

        x(i) = 1
    sfârșit pentru
    a = n - l1 + 2
    b = n
    Return
    altfel
    x(a - 1) = 1
    x(a) = 0
    b = a - 1
    pentru i = a - 1, 1, -1
        dacă x(i) = 0 atunci
            a = i + 1
            Return
        sfârșit dacă
    a = 1
    sfârșit pentru
sfârșit dacă{a < b}
Return

```

### Generarea mulțimii permutărilor

Complexitatea oricărui algoritm care rezolvă această problemă este de  $\Omega(n!)$ , deoarece trebuie generați  $n!$  vectori. Vectorul  $p$  cu  $n$  elemente va conține permutarea mulțimii  $A = \{1, 2, \dots, n\}$ . Există mulți algoritmi pentru rezolvarea acestei probleme, dar noi prezentăm aici numai doi dintre ei.

Prima metodă este: urmărește generarea vectorului  $x$  în ordine lexicografică crescătoare. Pentru construcția lui  $p'$  – succesorul lui  $p$  – vom pune în evidență ultima secvență descrescătoare:  $p(i) < p(i+1) > p(i+2) > \dots > p(n)$  și în consecință componenta care trebuie modificată este  $p(i)$ . În locul lui  $p(i)$  trebuie pusă o componentă  $p(j)$  cu  $j > i$ , cu condiția 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:  $(p(1), \dots, p(i-1), p(k), p(i+1), \dots, p(k-1), p(i), p(k+1), \dots, p(n))$  unde  $p(i+1) > p(i+2) > \dots > p(k-1)$ . Dacă  $p(i)$  nu ar fi mai mare decât  $p(k+1)$ , atunci am avea (datorită inegalității  $p(k) > p(k+1)$ ) că  $p(k)$  nu ar fi cel mai mic din secvența  $p(i+1), \dots, p(n)$  care este mai mare decât  $p(i)$ , contradicție cu alegerea lui  $k$ ; deci avem că  $p(i) > p(k+1)$ , deci  $p(i) > p(k+1) > \dots > p(n)$ . Ușor se demonstrează că  $p(k-1) > p(i)$ .

Rezultă că secvența de la  $p(i+1)$  la  $p(n)$  este descrescătoare și pentru a obține  $p'$  nu avem decât să inversăm această secvență, interschimbând termenii de la început cu cei de la sfârșit până la mijlocul secvenței. Algoritmul este:

```

Perm1(p, n, ig)
    dacă ig = 0 atunci
        pentru i = 1, n

```



```

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

```

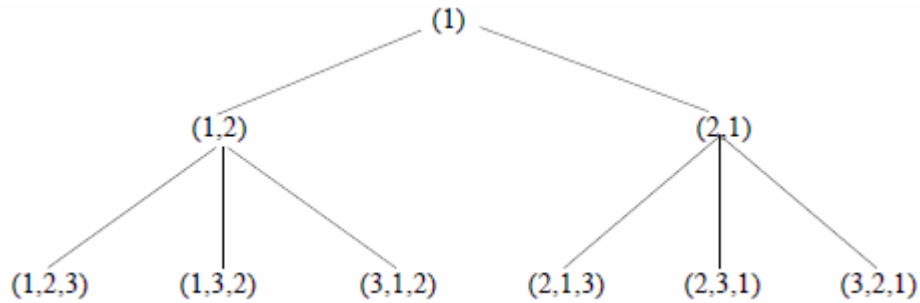
Metoda pe care o prezentăm în continuare se bazează pe generarea recursivă a permutărilor. Dacă avem  $S_k$  mulțimea permutărilor de  $k$  elemente de forma  $p = (p(1), \dots, p(k)) \in S_k$ , atunci  $S_{k+1}$  va conține pe baza fiecărei permutări  $p$  câte  $k+1$  permutări  $p'$ :  $(p(1), \dots, p(k), k+1)$ ,  $(p(1), \dots, p(k-1), k+1, p(k))$ ,  $\dots$ ,  $(p(1), k+1, p(2), \dots, p(k))$ ,  $(k+1, p(1), \dots, p(k))$ .

Deci permutările  $S_n$  vor fi frunzele unui arbore construit astfel:

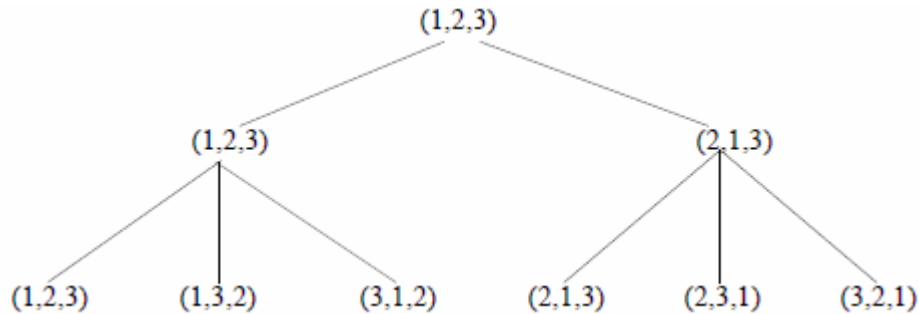
- rădăcina conține 1;
  - descendenții unei permutări  $p = (p(1), \dots, p(k))$  sunt cele  $k+1$  permutări descrise mai sus. Putem atașa acestui arbore unul în care toate permutările sunt de  $n$  elemente, prin completarea permutării de  $k$  elemente cu componentele  $(k+1, k+2, \dots, n)$ .
- Notăția  $a \leftrightarrow b$  înseamnă “interschimbă valoarea lui  $a$  cu valoarea lui  $b$ ”, care se efectuează prin următoarea secvență:  $aux = a, a = b, b = aux$ .



Exemplu: pentru  $n = 3$  se obține arborele din Figura 7.1 și arborele atașat din figură:



Arborele atașat unei permutări de trei elemente



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

Fie  $p = (p(1), \dots, p(n))$ ; atunci  $p_$  se construiește astfel: Dacă  $n$  apare pe o poziție  $m > 1$ , atunci se face interschimbarea  $p(m) \leftrightarrow p(m-1)$  și s-a obținut noua permutare. Dacă  $p(1) = n$ , atunci se permută circular spre stânga cele  $n$  elemente, obținându-se tatăl permutării  $p$ . În continuare, dacă  $n-1$  apare pe o poziție  $m \neq 1$ , atunci se efectuează interschimbarea  $p(m) \leftrightarrow p(m-1)$  și s-a obținut noua permutare; în caz contrar, se permută circular ultimele  $n-1$  poziții și se încearcă deplasarea spre stânga a lui  $n-2$ , etc. Algoritmul este:

Perm2( $p, n, ig$ )

dacă  $ig = 0$  atunci

pentru  $i = 1, n$

$p(i) = i$

sfârșit pentru

$ig = 1$

Return

sfârșit dacă

pentru  $i = n, 2, -1$

$m = 1$

cât timp  $p(m) \neq i$

$m = m + 1$

sfârșit cât timp

dacă  $m = 1$  atunci  
 $p(m - 1) \leftrightarrow p(m)$   
Return  
sfârșit dacă  
cheamă PermCircStanga( $p, n, i$ )  
 {se permută circular primele  $i$  componente ale lui  $P$ }  
Return  
sfârșit pentru  
 $ig = 0$   
Return  
 Unde PermCircStanga( $p, n, i$ ) este următorul algoritm:  
 PermCirc( $p, n, i$ )  
 $x = p(1)$   
pentru  $j = 1, i - 1$   
 $p(j) = p(j + 1)$   
sfârșit pentru  
 $p(i) = x$   
Return

### **Generarea mulțimii aranjamentelor**

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

Aranjl( $x, n, m, ig, c, p, ig1$ )

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

```

    pentru  $i = 1, m$ 
         $p(i) = i$ 
         $x(i) = c(i)$ 
    sfârșit pentru
     $ig1 = 1$ 
    Return
sfârșit dacă
sfârșit dacă
Return

```

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

Fie  $x = (x(1), \dots, x(m))$  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  $x(i)+1, x(i)+2, \dots, n$  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 găsește valoarea  $x$  cu proprietatea menționată,  $x(i), x(i+1), \dots, x(m)$  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. Obținem în acest fel următorul algoritm:

```

Aranj2( $x, n, m, ig, disp$ )
dacă  $ig = 0$  atunci
    pentru  $i = 1, m$ 
         $x(i) = i$ 
         $disp(i) = 1$ 
    sfârșit pentru
    pentru  $i = m + 1, n$ 
         $disp(i) = 0$ 
    sfârșit pentru
     $ig = 1$ 
    Return
sfârșit 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  $disp(k) = 0$   
          $x(l) = k;$   
          $disp(k) = 1$   
     sfârșit pentru  
     Return  
sfârșit dacă  
sfârșit pentru  
sfârșit pentru  
 $ig = 0$   
Return



### Rezumat

Pentru determinarea unor submulțimi (de o natură particulară) a unei mulțimi există algoritmi specializați. Se pot da, evident, mai mulți algoritmi pentru generarea unei soluții. Aceste tipuri de probleme se întâlnesc adeseori sub o formă mascată în enunțul altor probleme, sau ca etapă intermediară într-un caz mai mare.



### M2.U2.3. Test de autoevaluare a cunoștințelor

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

### **M2.U3. Metoda Divide et Impera**



#### **M2.U3.1. Introducere**

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 aceeași natură cu problema inițială), iar apoi prin recombinarea soluțiilor obținute. Determinarea complexității se reduce la rezolvarea unei recurențe, pentru care o unealtă utilă este Teorema centrală. Este dat cel mai eficient algoritm de sortare (Quicksort) și se demonstrează o limită inferioară pentru algoritmi de sortare bazați pe comparații.



#### **M2.U3.2. Obiectivele unității de învățare**

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- să folosiți metoda Divide et Impera
- să calculați complexitatea algoritmilor elaborați prin această metodă



**Durata medie de parcurgere a acestei unități de învățare este de trei ore.**

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

1. Partiționarea problemei inițiale 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 puțin performantă pentru cazuri de dimensiune mare;
3. Combinarea rezultatelor parțiale pentru a obține rezultatul problemei inițiale. Dacă o problemă a fost divizată în  $a$  subprobleme care au dimensiunea  $n/b$  și sunt rezolvate recursiv, atunci complexitatea este dată de formula:  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  unde  $f(n)$  este costul combinării soluțiilor subproblemelor.

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

**Teorema 1. (Teorema centrală.)** Fie  $a \geq 1$  și  $b > 1$  constante, fie  $f(n)$  o funcție și  $T(n)$  definită pe întregii nenegativi prin recurența:  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  unde interpretăm pe  $\frac{n}{b}$  fie ca pe

$\left\lceil \frac{n}{b} \right\rceil$ , fie ac pe  $\left\lfloor \frac{n}{b} \right\rfloor$ . atunci  $T(n)$  poate fi delimitat asimptotic după cum urmează:

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

Am folosit *divide et impera* la sortarea unui șir (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 inserție, care era  $T(n) = \Theta(n^2)$

Metoda constă în următoarele etape:

1. Partiționarea problemei inițiale 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 puțin performantă pentru cazuri de dimensiune mare;
3. Combinarea rezultatelor parțiale pentru a obține rezultatul problemei inițiale.

Am folosit *divide et impera* la sortarea unui șir (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 inserție, care era  $T(n) = \Theta(n^2)$  unde  $f(n)$  este costul combinării soluțiilor subproblemelor.

Am folosit *divide et impera* la sortarea unui șir (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 inserție, care era  $T(n) = \Theta(n^2)$

### 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 și trei baghete.

După o străveche legendă hindusă, zeul Brahma ar fi fixat pe masa templului său din Benares trei bastonașe verticale de diamant și ar fi înșirat 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 și al treilea bastonaș; zi și noapte trebuia mutat câte unul din discurile aflate în vârful unuia din bastonașe în vârful altui bastonaș, respectând în fiecare moment ordinea descrescătoare a mărimii discurilor aflate pe fiecare bastonaș. Legenda spune

că, dând această poruncă, Brahma a susținut că atunci când se va termina așezarea celor 64 de dicuri pe cel de-al doilea baștonaș, atunci se va sfârși și viața 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 bastonaș în circa 584.942.416.729 de ani.

Enunțul formalizat al problemei este: fie trei tije și  $n$  discuri perforate de diametre diferite. Discurile sunt așezate inițial 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, ținând cont de următoarele reguli:

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

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

Să demonstrăm valabilitatea acestui rezultat prin inducție matematică completă după  $n$ . Am arătat că afirmația 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 doua 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.  $2n - 1$  mutări pentru realizarea celei de a treia etape

În total sunt suficiente  $2^{n+1} - 1$  mutări.

### Metoda 1

O mutare poate fi scrisă ca o pereche  $(i, j)$  cu  $i \in \{1, 2, 3\}$  și  $i \neq j$ , semnificând că se mută discul cel mai mic de pe tija  $i$  pe tija  $j$ . Mai notăm cu  $H(m, i, j)$  șirul 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 șirul de mutări  $H(m, i, j)$ . Observăm că putem scrie:

$$H(m, i, j) = \begin{cases} (i, j) & \text{pentru } m = 1 \\ H(m-1, i, k), (i, j), H(m-1, k, j) & \text{pentru } m \neq 1 \end{cases}, \text{ unde } k = 6 - i - j \text{ este}$$

tija diferită de tijele  $i$  și  $j$ .

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 obține cele  $2^n - 1$  coloane, fiecare



coloană memorând o mutare. Începem prin a înscrie perechea (1,2) pe coloana  $2^{n-1}$ . Perechile ce vor lua naștere din ea vor fi înscrise pe coloanele  $2^{n-2}$  și  $2^{n-1} + 2^{n-2}$ . Se observă că acest algoritm face parte din clasa algoritmilor “Divide et Impera”.

Mai general, pentru fiecare valoare a lui  $m \in \{n, n-1, \dots, 2\}$  se expandează perechile din coloanele  $c$  de forma  $c = M(2^m) + 2^{m-1}$ , perechile care rezultă din cea din coloana  $c$  fiind înscrise pe coloanele  $c - 2^{m-2}$ ,  $c + 2^{m-2}$ . ( $Mp$  este notație pentru “multiplu de  $p$ ”)

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  $l = k2, N, k1$

$i = M(1, l)$

$j = M(2, l)$

$k = 6 - i - j$

$M(1, l - k3) = i$

$M(2, l - k3) = k$

$M(1, l + k3) = k$

$M(2, l + k3) = j$

sfârșit pentru

$k1 = k2$

$k2 = k3$

$k3 = k3/2$

sfârșit pentru

return

### **Algoritmul de sortare QuickSort.**

Fie șirul  $x = (x(1), x(2), \dots, x(n))$ . Dorim să sortăm acest șir, adică  $\forall i \in \{1, \dots, n-1\}$ ,  $x(i) \leq x(i+1)$ . După cum știm, algoritmii sunt caracterizați de timpul de execuție și spațiul de memorie ocupat. Sortarea unui șir cu metoda bulelor sau cea a inserției s-a făcut cu algoritmi “in place” (în afara șirului s-au menținut cel mult un număr constant de elemente din șir), iar prin metoda interclasării se ocupă un alt șir (nu este un algoritm “in place”) și în plus timpul de execuție ascunde o constantă multiplicativă mare.

Ne dorim să realizăm un algoritm de sortare prin metoda “Divide et Impera”, fără a folosi un șir intermediar, deci “in place”. Profesorul C.A.R. Hoare de la Universitatea Oxford a propus împărțirea șirului în părți nu neapărat egale având proprietatea că toate elementele primei părți 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 părți să fie mai mari decât  $x(k)$ . Faza de combinare lipsește (este o metodă “Divide et Impera” șchioapă).

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( $x, k + 1, r$ )

sfârșit dacă

Return

Algoritmul care realizează partiționarea 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

$k = j$

$terminat = adevarat$

sfârșit dacă

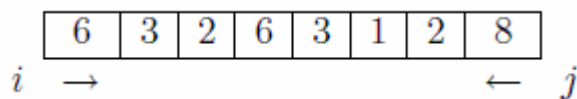
până când  $terminat = adevarat$

Return



### Exemplu

Fie șirul



Avem  $pivot=6$ ,  $p=1$ ,  $r=8$ . Primul element al șirului mai mic sau egal decât pivotul este 2 (în căutarea de la dreapta la stânga) și primul element al șirului mai mare sau egal decât pivotul este 6 (în căutarea de la stânga la dreapta). După găsirea lor le interschimbăm, ajungând la configurația:

2	3	2	6	3	1	6	8
$i$						$j$	

Apoi succesiv:



2	3	2	1	3	6	6	8
			$i$			$j$	

2	3	2	1	3	6	6	8
				$j$	$i$		

Deoarece  $j$  a devenit mai mic decât  $i$ , se iese din procedura de partiționare cu valoarea lui  $k=5$ . În acest moment s-a terminat partiționarea cu  $pivot=6$ . În continuare se face partiționarea șirului din partea stângă pentru  $pivot=2$ :

2	3	2	1	3
$i$				$j$

1	3	2	2	3
$i$			$j$	

1	2	3	2	3
	$i$		$j$	

1	2	3	2	3
		$j$	$i$	



Aplicați algoritmul de mai sus pentru șirul 10,9,3,4,5,2,7.

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) sortați.

*Observație:* Când șirul este deja ordonat crescător, se face partiționare după fiecare element.

Complexitatea partiționării este dată de  $T_P(n) = c \cdot n$ ,  $c$  constantă. Complexitatea QuickSort în cazul șirului inițial sortat este:

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

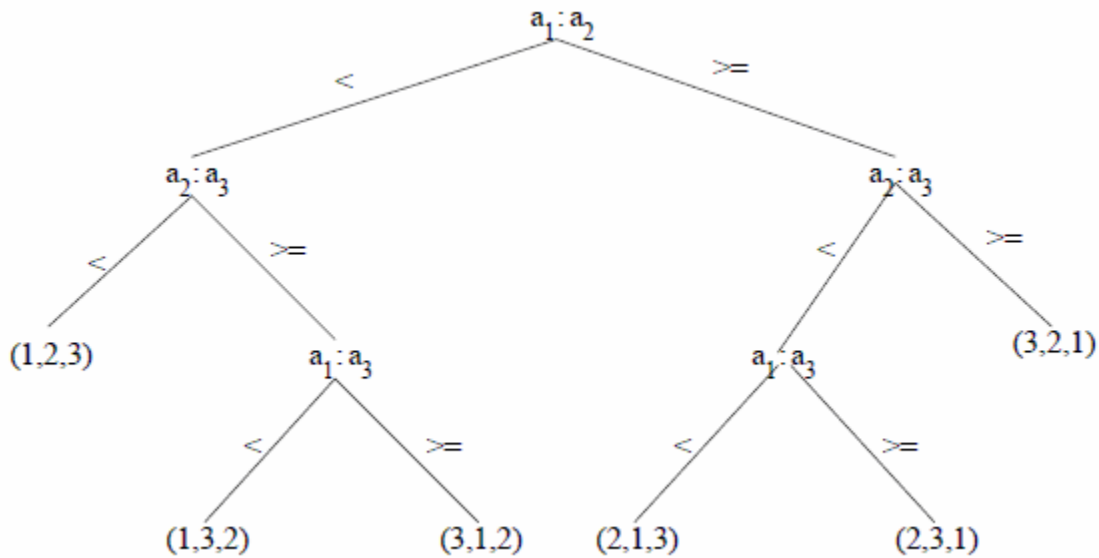
unde am considerat în mod convenabil că  $T_Q(1)$  este mărginit de aceeași constantă  $c$  care apare în expresia lui  $T_P(n)$ . În cel mai bun caz, când șirul se împarte în jumătate,  $T_Q(n) = 2T_Q(n/2) + \Theta(n) = \Theta(n \log n)$  (conform teoremei centrale). Se poate demonstra că timpul mediu al QuickSort-ului este  $\Theta(n \log n)$ . În practică se dovedește 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 comparații 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 următoare). Se obține un arbore binar care are drept frunze permutările indicilor elementelor inițiale, adică  $3!$  frunze. În cazul a  $n$  valori, obținem un arbore binar cu  $n!$  frunze. Prin înălțimea unui arbore înțelegem lungimea drumului maxim de la rădăcină la orice frunză și se notează cu  $h$ .  $h$  reprezintă numărul de comparații 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 că  $h \geq \log_2(n!)$  și folosind aproximația lui

$$\text{Stirling } n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ obținem că } h \geq \log_2 n! > n \log_2 n - n \log_2 e = \Omega(n \log_2 n)$$

Se obține că  $T(n) = \Omega(n \log_2 n)$ . Am demonstrat deci că pentru algoritmii care se bazează doar pe comparații complexitatea nu poate să scadă sub  $n \log_2 n$ .

În figura de mai jos se poate vedea arborele de decizie pentru trei elemente





## Rezumat

Metoda Divide et Impera se folosește atunci când rezolvarea unei probleme se poate face prin descompunerea ei în cazuri de dimensiune mai mică, rezolvarea acestora după același principiu, apoi combinarea rezultatelor lor. Deseori algoritmi de acest tip se scriu recursiv (datorită naturii lor), iar determinarea complexității se face prin rezolvarea unei ecuații recursive, a cărei soluții se poate da de multe ori pe baza Teoremei centrale. Un exemplu important de algoritm de tip Divide et Impera este Quicksort.



### M2.U3.3. Test de autoevaluare a cunoștințelor

1. Bazându-vă pe observația că mutarea din mijloc se știe, scrieți un algoritm iterativ pentru problema turnurilor din Hanoi, punând mereu mijloacele intervalelor obținute prin înjumătățire într-un șir cu numărul cunoscut de mutări.
2. Având la intrare un șir ordonat crescător, să se scrie un algoritm de tip “Divide et Impera” pentru găsirea poziției unui element de valoare dată  $x$ , dacă el există. Se va folosi metoda înjumătățirii intervalului dat de numărul de elemente al șirului. Care este complexitatea căutării?
3. Rezolvați aceeași problemă prin căutare secvențială (testarea pe rând a elementelor). Care este în acest caz complexitatea?
4. Demonstrați că timpul de execuție 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, partiționarea se face cu un singur element într-una din partiții. Rescrieți algoritmul de partiționare, luând ca element de separare nu primul element, ci valoarea din mijloc între primul element, ultimul și cel aflat pe poziția din mijloc.

## M2.U4. Sortare și statistici de ordine



### M2.U4.1. Introducere

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

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

*Observație:* Vom considera că șirul este cu elemente distincte în rezolvarea tuturor problemelor enunțate, urmând apoi să extindem rezultatele obținute și pentru situația în care elementele se repetă. Vor fi considerați mai mulți algoritmi care rezolvă această problemă.

Unitatea de învățare conține și prezentări ale altor algoritmi de sortare (heapsort, cu care ocazie se introduce și structura de date numită heap), sortări în timp liniar.



### M2.U4.2. Obiectivele unității de învățare

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- folosească diverși algoritmi de sortare
- scrie algoritmi pentru probleme de statistică de ordine



**Durata medie de parcurgere a acestei unități de învățare este de trei ore.**



### Exemplu

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



Aplicați exemplul de mai sus pentru șirul 10, 9, 3, 4, 5, 2, 7.

Mediana depinde de paritatea numărului de elemente din șirul dat. Dacă  $n$  este impar atunci mediana este unică și se găsește în șirul sortat pe poziția  $i = (n + 1)/2$ . Altfel avem de-a

face cu două mediane, una dintre ele pe poziția  $i = n/2$ , cealaltă pe poziția  $i = n/2 + 1$ . În general, pentru un șir de lungime  $n$  vom considera mediană elementul de pe poziția  $(n+1)/2$  din șirul sortat. În cazul nostru mediana este 5.

Ce reprezintă minimul și maximul din perspectiva statisticilor de ordine?

Minimul este prima componentă în ordine statistică a unui șir. Maximul este a  $n$ -a componentă în ordine statistică.

### **Determinarea simultană a minimului și a maximului**

MinMax( $x, n$ )

dacă  $n \bmod 2 = 0$  atunci

dacă  $x(1) < x(2)$

atunci

$min = x(1)$

$max = x(2)$

altfel

$min = x(2)$

$max = x(1)$

sfârșit dacă

$k = 3$

altfel

$min = x(1)$

$max = x(1)$

$k = 2$

sfârșit dacă

pentru  $i = k, n - 1, 2$

dacă  $x(i) < x(i + 1)$

atunci

$mic = x(i)$

$mare = x(i + 1)$

altfel

$mic = x(i + 1)$

$mare = x(i)$

sfârșit dacă

dacă  $mic < min$

atunci

$min = mic$

sfârșit dacă

dacă  $mare > max$

atunci

$max = mare$

sfârșit dacă

sfârșit pentru

scrie( “minimul este ”, *min* )

scrie( “maximul este ”, *max* )

Return

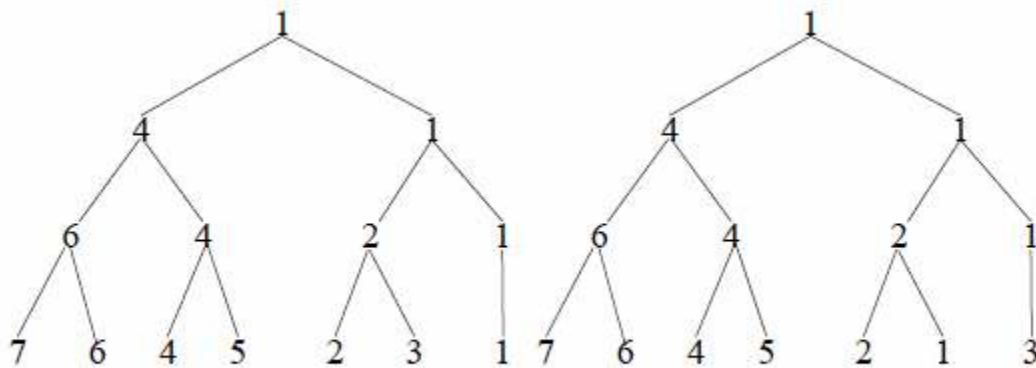
Numărul de comparații între elemente necesare în determinarea independentă a *min* și *max* este  $2(n-1)$  (prin două parcurgeri ale șirului), în schimb pentru determinarea simultană a *min* și *max* avem nevoie de  $3\lceil n/2 \rceil - 2$  comparații (demonstrați acest lucru).



Sugerăm cititorului să execute algoritmul pentru intrarea  $x = (1, 2, -3, 4, 5, 6, -85, 2)$ .

### Găsirea celor mai mici două elemente dintr-un șir

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 “câștigătoare” se vor compara la rândul lor pe perechi adiacente, ca mai sus. Când rămâne un singur element, acesta este minimul șirului. Atașăm acestor operații un arbore binar (posibil nestructurat) echilibrat, construit în felul următor: există  $n$  noduri terminale (frunze), fiecare conținând drept informație câte un element din șir; un nod neterminal are drept valoare minimul din cei doi fii.



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

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

Evident că pentru fiecare nod neterminal, informația 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 inițial “învins” de minim: în exemplele date este 2, “învins” de 1. Pentru aceasta vom considera informația din cei doi descendenți ai rădăcinii, în ambele exemple date fiind 4 și 1. 4 este cel mai mic element din jumătatea stângă a șirului (este descendent stâng al rădăcinii) și va trebui să căutăm în jumătatea dreaptă a șirului cel mai mic număr, învins de rădăcină; acest lucru se face (în exemplele date) coborând în subarboarele drepte spre frunze. Se compară toate elementele pe care informația din rădăcină le-a depășit cu minimul din jumătatea stângă (în cazul nostru 4); coborârea se termină atunci



când fie se ajunge la frunze (cazul (b)), fie este evident că nu mai are sens coborârea (cazul (a)). În algoritmul de mai jos nu se va construi explicit arborele. Se vor folosi niște cozi pentru menținerea pentru fiecare element din șir a elementelor “învins” de acesta.

MinMin( $x, n, min1, min2$ )

pentru  $i = 1, n$

$C(i) = \emptyset$  { $C(i)$  este coadă}

sfârșit 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

$imin = i + 1$

$imax = i$

sfârșit dacă

$a(pozitie) = a(imin)$

$C(pozitie) = C(imin) \cup \{a(imax)\}$   $pozitie = pozitie + 1$

dacă  $n \bmod 2 = 0$  atunci

{a rămas un element care nu are cu cine se compara}

$a(pozitie) = a(n)$

$C(pozitie) = C(n)$

$pozitie = pozitie + 1$

sfârșit dacă

$n = pozitie - 1$

sfârșit pentru

sfârșit cât timp

$min1 = a(1)$

$min2 \leftarrow C(1)$

cât timp  $C(1) \neq \emptyset$

$y \leftarrow C(1)$

dacă  $y < min2$  atunci

$min2 = y$

sfârșit dacă

sfârșit cât timp

Return

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

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

### Selecție în timp mediu liniar

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

Sel( $A, p, r, i, rez$ )

dacă  $p = r$  atunci

$rez = a(p)$

Return

sfârșit dacă

Part( $A, p, r, k$ ) {Part este algoritmul de partiționare din QuickSort}

dacă  $i \leq k$  atunci

Sel( $a, p, k, i, rez$ )

altfel

Sel( $a, k + 1, r, i - k, rez$ )

sfârșit dacă

Return

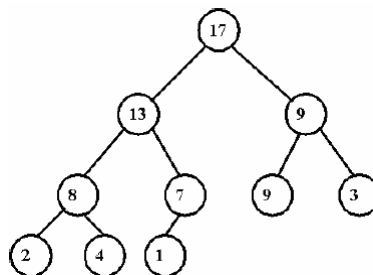
Algoritmul Sel aplicat unui șir deja sortat are (cel mai rău caz) complexitatea  $O(n^2)$ . Se poate arăta că pentru cazul mediu complexitatea este de  $O(n)$ .

### Heapsort

Fie vectorul:

17	13	9	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Arborele atașat este cel din figura de mai jos. Acest arbore are adâncimea de  $\log_2 n$ .



O structură careia i se poate pune în corespondență un arbore binar echilibrat se numește 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 numește 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  $l \leq n$  și  $a(l) > a(i)$  atunci

$imax = l$

sfârșit dacă

dacă  $r \leq n$  și  $a(r) > a(imax)$  atunci

$imax = r$

sfârșit dacă

dacă  $i = imax$  atunci

$a(i) \leftrightarrow a(imax)$

sfârșit dacă

până când  $i = imax$

Return

Algoritmul pentru construcția unui heap este următorul:

ConstruiesteHeap( $a, n$ )

pentru  $i = \lfloor n/2 \rfloor, 1, -1$

Heapify( $a, n, i$ )

sfârșit pentru

Return

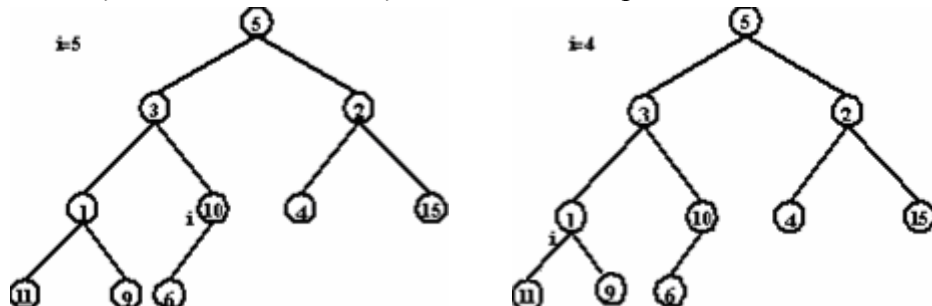


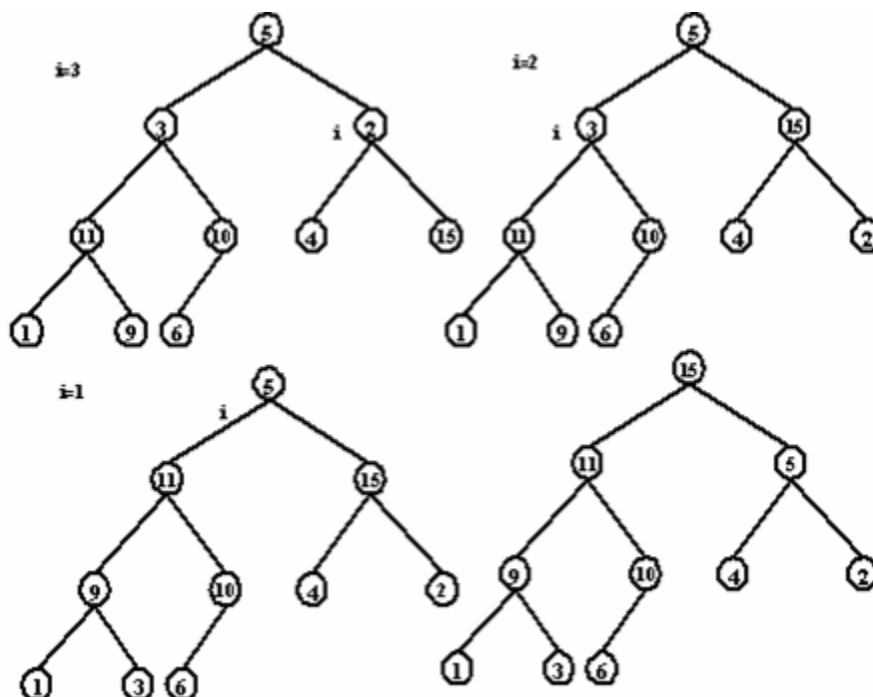
### Exemplu

Fie șirul

$a =$	5	3	2	1	10	4	15	11	9	6
	1	2	3	4	5	6	7	8	9	10

Urmăriți, în continuare evoluția arborelui la heapificare.





Aplicați exemplul de mai sus pentru șirul 10, 9, 3, 4, 5, 2, 7.

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

Deci timpul de execuție pentru ConstruiesteHeap este:

$$T(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor, \quad O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) \text{ dar } \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

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

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

HeapSort( $a, n$ )

Cheama ConstruiesteHeap( $a, n$ )

pentru  $i = n, 2, -1$

$a(1) \leftrightarrow a(i)$

Heapify( $a, i - 1, 1$ )

sfârșit pentru

Return

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

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 ținerea în evidență dinamică a

cozilor de prioritați unde o complexitate de  $n$  care s-ar putea obține printr-un algoritm cu inserție este înlocuită cu o complexitate de  $\log n$  păstrând coada într-un heap.

### 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 situații speciale ale șirului. Astfel, dacă șirul este constituit din numere întregi pozitive (sau măcar întregi), sau dacă ele sunt numere distribuite uniform într-un interval, sau luând în considerare cifrele numerelor din șir se pot da diferiți algoritmi de complexitate liniară pentru sortare.

### Algoritmul sortării prin numărare (CountSort).

Un astfel de algoritm este algoritmul de sortare prin numărare numit CountSort. Fie  $(x(i))_{i=1,n}$  un șir de numere naturale strict pozitive și  $k$  maximul acestui șir. Sortarea prin numărare constă în găsirea a câte elemente sunt mai mici decât  $x(i)$  (pentru fiecare  $i$ ); în acest fel vom ști pe ce poziție trebuie să se afle  $x(i)$  în șirul sortat (dacă sunt  $m$  elemente mai mici decât  $x(i)$ , atunci el ar trebui să se afle pe poziția  $m + 1$  în șirul sortat). Considerăm șirul inițial  $x = (x(1), x(2), \dots, x(n))$ ;  $y = (y(1), y(2), \dots, y(n))$  va fi șirul sortat, iar  $c = (c(1), c(2), \dots, c(k))$  este un șir ajutător.

CountSort( $x, n, y, k$ )

pentru  $i = 1, k$

$c(i) = 0$

sfârșit pentru

pentru  $j = 1, n$

$c(x(j)) = c(x(j)) + 1$

sfârșit pentru

pentru  $i = 2, k$

$c(i) = c(i) + c(i - 1)$

sfârșit pentru

pentru  $j = n, 1, -1$

$y(c(x(j))) = x(j)$

$c(x(j)) = c(x(j)) - 1$

sfârșit pentru

Return



### Exemplu

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

- elementele lui  $c$  sunt inițializate cu 0:  $c = (0, 0, 0, 0, 0, 0)$

- pentru  $i$  de la 1 la  $k$ , pe poziția  $i$  din  $c$  obținem numărul de elemente egale cu  $i$  în șirul  $x$ :  $c = (2, 0, 2, 3, 0, 1)$

- pentru  $i = 2, k$ , pe poziția  $i$  a șirului  $c$  se obține câte elemente mai mici sau egale cu  $i$  sunt în șirul  $x$ :  $c = (2, 2, 4, 7, 7, 8)$

- pentru  $j = n, 1$ , cu pasul -1, se află componentele șirului  $y$  sortat:  $y = (1, 1, 3, 3, 4, 4, 4, 4, 6)$



Aplicați exemplul de mai sus pentru șirul 10,9,3,4,5,2,7.

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  $\Theta(n)$ .

### Sortare pe baza cifrelor

Să presupunem că avem un șir de numere naturale, având toate același număr  $d$  de cifre (dacă nu, atunci un număr se poate completa cu zerouri esemnificative până când atinge numărul maxim de cifre din șir). Se poate face ordonarea șirului de numere în felul următor: se pornește de la cifra cea mai puțin semnificativă (ultima cifră) și se sortează elementele pe baza acestei informații, obținându-se o altă ordine (un alt șir). Se pornește de al acest ultim șir generat și se sortează elementele după următoarea cifră mai puțin semnificativă (penultima cifră), pentru care se face sortarea elementelor; și așa mai departe, în final făcându-se sortarea după cea mai semnificativă cifră (prima cifră). Esențial 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 mai puțin semnificativă), atunci se pleacă de la ordinea obținută 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 șir  $x = (x(1), \dots, x(n))$  avem că  $x(i) = x(j)$  și  $i < j$ , prin sortare se ajunge ca  $x(i)$  să fie repartizat pe poziția  $i_-$ ,  $x(j)$  să fie repartizat pe poziția  $j_-$ , 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  $i$

sfârșit 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 secțiunea 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 și  $b = O(n)$ , atunci complexitatea este liniară.



### Rezumat

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



### **M2.U4.3. Test de autoevaluare a cunoștințelor**

1. Descrieți un algoritm care, fiind dată o mulțime  $S$  de  $n$  numere întregi și distincte și un întreg pozitiv  $k \leq n$ , determină cele  $k$  numere care sunt cele mai apropiate de mediana lui  $S$ .
2. Fie  $a[1 \dots n]$ ,  $b[1 \dots n]$  două șiruri sortate. Scrieți un algoritm performant care găsește mediana celor  $2n$  numere.
3. Fiind dată o mulțime de  $n$  numere, dorim să găsim cele mai mari  $i$  numere în ordinea sortată, folosind un algoritm bazat pe comparații. Să se scrie mai multe variante și să se compare din punct de vedere al complexității.
4. Să se modifice algoritmul CountSort astfel încât să sorteze și șiruri conținând numere întregi negative.

## M2.U5. Metoda Greedy



### M2.U5.1. Introducere

Unitatea introduce o nouă metodă de rezolvare a problemelor. Ea va arăta cum se abordează problemele pentru care soluția se poate construi prin alegeri ale componentelor, alegeri asupra cărora nu se revine. Se dorește 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 enunțate și rezolvate (sau sunt făcute trimiteri bibliografice).



### M2.U3.2. Obiectivele unității de învățare

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- abordeze probleme pentru care soluția se poate construi prin alegeri ale componentelor, alegeri asupra cărora nu se revine
- nu scrie algoritmi Greedy fără demonstrația corectitudinii



**Durata medie de parcurgere a acestei unități de învățare este de două ore.**

Metoda Greedy este o metodă de elaborare a algoritmilor ce determină  $x^* \in X$  ce verifică o condiție dată  $C$ . Exemplul semnificativ este cel al problemelor de optimizare în care se determină  $x^*$  ce minimizează sau maximizează o funcție obiectiv. Să presupunem că soluția  $x^*$  are mai multe componente și că spațiul soluțiilor  $X$  este cunoscut. De asemenea presupunem că  $C$  oferă un criteriu de selecție prin care se pot alege valori pentru componentele soluției. Modul de rezolvare al problemelor dat de strategia Greedy este acela de construcție a soluției componentă cu componentă. Se pleacă de la soluția vidă și la fiecare pas al algoritmului se selectează o valoare pentru o componentă; dacă această valoare convine, atunci aceasta este pusă în soluție pe componenta respectivă. Într-un cuvânt, *se ia tot ce este bun* pentru construcția soluției. Această selecție se face în funcție 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 selecție. Dacă la fiecare pas alegem cea mai bună valoare în spiritul lui  $C$ , nu este necesar ca în final să obținem soluția optimă căutată. Este ceea ce se întâmplă în problemele de optimizare: o succesiune de creșteri poate să ducă la un optim local și nu la cel global căutat. Dacă se reușește să se demonstreze că soluția găsită verifică condiția  $C$  atunci am obținut o soluție pentru problemă. În caz contrar, dacă soluția găsită este suficient



de aproape de soluția optimă spunem că avem o *soluție Greedy euristică*. Cele prezentate anterior pot fi descrise de procedura Greedy de mai jos:

Greedy( $x$ )

$x = \emptyset$

cât timp soluția nu este completă

    selectează o valoare  $a$

dacă valoarea  $a$  convine

atunci

$x = x \cup \{a\}$

sfârșit dacă

sfârșit cât timp

Return

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

1. În funcție de condiția  $C$  se găsește un criteriu de selecție a valorilor
2. Se elaborează algoritmul respectând schema anterioară
3. Se demonstrează corectitudinea alegerii făcute arătând că soluția  $x^*$  respectă  $C$ . În caz contrar se încearcă să se determine cât de departe este soluția găsită de algoritm față de soluția căutată.

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

Exemplificăm metoda anterioară prin câteva probleme clasice.

### Submulțimi de sumă maximă

Enunțul problemei: Dacă  $x = (x(i), i = 1, n)$  reprezintă o mulțime de elemente atunci să se determine submulțimea  $y = (y(i), i = 1, k)$  de sumă maximă. Date de intrare:

$n$  - dimensiunea vectorului

$x = (x(i), i = 1, n)$ , mulțimea de elemente;

Date de ieșire:

-  $k$  numărul de elemente al submulțimii  $y = (y(i), i = 1, k)$

- submulțimea de elemente.

Problema anterioară este o problemă clasică de optimizare care se rezolvă prin Greedy.

Submulțimea căutată este o soluție a problemei de optimizare  $\max \left\{ \sum_{y \in Y} y \mid \Phi \neq Y \subseteq X \right\}$

Simplitatea problemei de optimizare oferă un criteriu de selecție foarte simplu bazat pe observațiile:

1. dacă în mulțimea  $x$  sunt numere pozitive, adăugarea unuia la submulțimea  $y$  conduce la creșterea sumei; găsim astfel submulțimea  $y$  formată din toate elementele pozitive;
2. în cazul în care toate elementele sunt negative, submulțimea  $y$  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

$k = k + 1$

$y(k) = x(i)$

sfârșit dacă

sfârșit pentru

dacă  $k = 0$

atunci

$max = x(1)$

pentru  $i = 2, n$

dacă  $x(i) > max$

atunci

$max = x(i)$

sfârșit dacă

sfârșit pentru

$k = 1$

$y(k) = max$

sfârșit dacă

Return

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

**Teorema de Corectitudine.** *Submulțimea  $y$  determinată de către algoritm are suma elementelor maximă față de toate submulțimile lui  $x$ .*

*Demonstrație.* Notăm  $sum(x)$  suma elementelor din  $x$ . Fie  $z$  submulțimea 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  $sum(z) = sum(y)$  și deci  $y$  este submulțime de sumă maximă.

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

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

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

Rămâne că  $z$  nu conține elemente negative și conține toate pozitivele, deci  $y = z$

## Arborele Huffman

Enunțul problemei: dacă  $S = \{s_1, s_2, \dots, s_n\}$  este o mulțime de semnale și  $p = (p(i), i = 1, n)$  este vectorul de probabilitate asociat lui  $S$ , atunci să se determine codul de lungime optimă asociat lui  $S$ .

Date de intrare:

- $n$  numărul de semnale,
- $p = (p(i), i = 1, n)$  vectorul de probabilități.

Date de ieșire -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 puține caractere, astfel încât costul transmisiei să fie cât mai mic. Înainte de a da rezolvarea problemei, este bine să facem o prezentare a câtorva noțiuni de teoria codurilor.

Un cod binar este o aplicație injectivă  $\varphi : S \rightarrow \{0, 1\}^+$ . Lungimea medie a codului în

funcție de sistemul de probabilități este definită de  $L(\varphi) = \sum_{i=1}^n |\Phi(s_i)| p(i)$ , unde ( $|a|$  este numărul de caractere a lui  $a$ ). Un cod este optim dacă minimizează lungimea medie. Dacă  $\varphi: S \rightarrow \{0, 1\}^+$  este un cod instantaneu, atunci arborele  $m$  - ar asociat se notează cu  $T_\varphi$  și verifică:

- are atâtea frunze câte semnale are  $S$ ;
- drumul în arbore de la rădăcină la frunză  $s \in S$  dă codificarea  $\varphi(s)$  prin concatenarea marcajelor muchiilor.

Lungimea medie a unui cod poate fi transportată într-o lungime medie a arborelui asociat, dată de  $L(\Phi) = \sum_{i=1}^n |\Phi(s_i)| p(i) = \sum_{i=1}^n \text{niv}_{T_\Phi}(s_i) p(i) = L(T_\Phi)$

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

Vom lucra cu structuri arborescente reprezentate înălțuit în care în câmpul de informație avem probabilitatea sau frecvența simbolului. Până în acest moment nu am spus nimic despre modul de aplicare al strategiei Greedy. Rămâne afirmația făcută inițial că semnalele cele mai probabile sunt codificate cu cât mai puține caractere.

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

- inițial pădurea este formată din vârfuri izolate reprezentând semnalele.
- se repetă o execuție dată de:
  1. se selectează arborii pădurii având informația cea mai mică;
  2. se conectează cei doi arbori aleși într-unul singur printr-o rădăcină comună având informația suma informațiilor 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}

$LS(s(i)) = *$

$LD(s(i)) = *$

$INFO(s(i)) = p(i)$

sfârșit pentru

Cheama 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  $j > 0$  și  $INFO(x) > INFO(s(j))$

$s(j + 1) = s(j)$

$j = j - 1$

sfârșit pentru

$s(j + 1) = x$

sfârșit 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 de complexitate.** Algoritmul *Huffman* are complexitatea  $\Theta(n^2)$ .

*Demonstrație.* Complexitatea algoritmului este dată de complexitatea sortării și de complexitatea selecțiilor repetate. Complexitatea sortării este  $\Theta(n \log n)$ . Selecțiile repetate ce se fac utilizează principiul inserării directe. Pentru a insera semnalul  $p$  în structura  $(s(j), j = 1, i-1)$  facem cel mult  $i-1$  comparații, găsind în final un număr de  $\sum_{i=1}^n i - 1 = \frac{n(n-1)}{2} = \Theta(n^2)$  comparații.

Rezultă complexitatea din enunț.

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

*Demonstrație.*

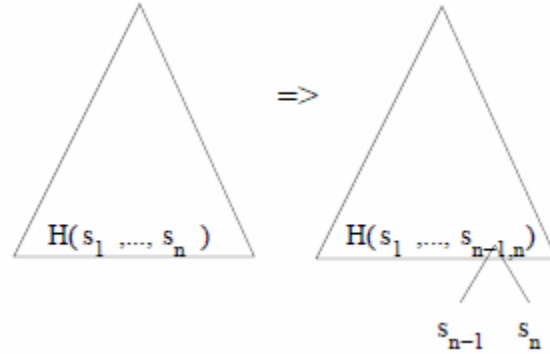
Vom folosi în demonstrație următoarele notații:

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

Despre arborele optim se poate arăta:

- $p_i < p_j$  rezultă  $nivT(s_i) > nivT(s_j)$
- există un arbore optim în care cele mai puțin probabile semnale sunt frați.

Să presupunem că  $s_i$  și  $s_j$  sunt semnalele cu probabilitatea cea mai mică și că prin unirea celor două semnale se creează un nou semnal notat  $s_{i,j}$ . Algoritmul Huffman construiește arborele binar după regula recursivă dată de figură. Avem următorul calcul de lungime:



Crearea unui nod pe baza semnalelor cele mai puțin probabile

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

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

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

în final vom avea

$$L(H(p(1), \dots, p(n))) = L(H(p(1), \dots, p(n-1) + p(n))) + p(n-1) + p(n)$$

Analog se arată că și arborele optim, care îi are ca frați pe  $s_{n-1}$  și  $s_n$  verifică

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

Demonstrația optimalității se va face prin inducție după  $n$ . Pentru  $n = 2$  algoritmul ne oferă  $H(p(1), p(2))$  care codifică pe  $s_1$  cu 0 și pe  $s_2$  cu 1, ceea ce înseamnă că este un cod optim.

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

Fie  $p = (p(i), i = 1, n+1)$  un sistem cu  $n+1$  probabilități în care  $p(n-1)$  și  $p(n-2)$  sunt cele mai mici două numere din șirul  $p$ . Aplicând formulele găsite anterior obținem:

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

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

Pentru că arborele Huffman codifică  $n$  semnale optim vom găsi:

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

Din relațiile de mai sus vom găsi:

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

și deci  $L(T(p(1), \dots, p(n), p(n+1))) \geq L(H(p(1), \dots, p(n), p(n+1)))$  Din optimalitatea lui  $T$  vom găsi:  $L(T(p(1), \dots, p(n), p(n+1))) = L(H(p(1), \dots, p(n), p(n+1)))$  deci  $H$  este un arbore optim.

### Interclasare optimă

Enunțul problemei: Dându-se o mulțime de vectori sortați crescător să se elaboreze un algoritm care determină interclasarea acestora cu numărul minim de comparații.

Date de intrare:

$n$  - numărul de vectori,

$nr = (nr(i), i = 1, n)$  - numărul de elemente din fiecare vector,  $x(i) = (x(i, j), j = 1, nr(i))$  șirul de vectori.

Date de ieșire:

$y = (y(i), i = 1,$

$nr(1) + \dots + nr(n))$ , 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 comparații 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 funcție de modul de interclasare al vectorilor de interclasare se obțin diferite soluții cu număr distinct de comparații. Este deci esențial să găsim o ordine care conduce către numărul minim de comparații.

Vom nota  $x+y$  vectorul obținut prin interclasarea lui  $x$  cu  $y$ .



### Exemplu

Dacă avem 4 vectori  $x, y, z, t$  cu respectiv 10, 20, 30 și 40 de elemente atunci:

- interclasarea  $((x+y)+z)+t$  se face cu  $(10+20-1)+(30+30-1)+(60+40-1)=187$  comparații,

- interclasarea  $((z+t)+y)+x$  se face cu  $(30+40-1)+(70+20-1)+(90+10-1)=257$  comparații,

deci prima strategie este mai bună decât a doua (de fapt, este chiar cea optimă).

Din acest exemplu se observă că maximă 40 se regăsește în toate interclasările ce se efectuează conducând la un număr mare de comparații.



Aplicați exemplul de mai sus pentru vectorii cu 15, 20, 4, 5, 7, 30 de componente.

Idea ar fi ca vectorii cu dimensiuni mici să 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 selecția să se facă cât mai ușor păstrăm 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 în matricea  $x$  folosind diferite tehnici: inserția directă sau inserția binară. Se repetă procesul până când rămâne un singur vector.

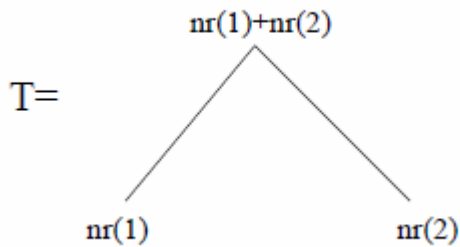
```

InterOpt( $n, nr, x, z$ )
Cheama Sortare( $n, nr, x$ )
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  $j > 0$  și  $n(j) > n1 + n2$ 
        pentru  $k = 1, nr(j)$ 
             $x(j + 1, k) = x(j, k)$ 
        sfârșit pentru
         $nr(j + 1) = nr(j)$ 
         $j = j - 1$ 
    sfârșit cât timp
    pentru  $k = 1, n1 + n2$ 
         $x(j + 1, k) = y(k)$ 
    sfârșit pentru
     $nr(j + 1) = n1 + n2$ 
sfârșit pentru
Return

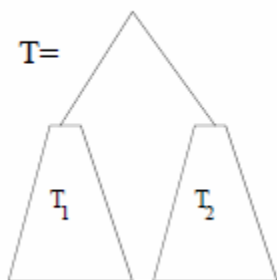
```

În procedura anterioară procedura *Sortare* determină rearanjarea elementelor (liniilor) matricii  $x$  descrescător în funcție 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 informație numărul de elemente ale vectorului;
2. arborele asociat interclasării a doi vectori  $x(1)$  și  $x(2)$  este reprezentat grafic în figură
3. Dacă  $T1$  este arborele asociat șirului de vectori  $x(i1), \dots, x(ik)$  iar  $T2$  este arborele asociat șirului de vectori  $x(ik+1), \dots, x(in)$  atunci  $T$  din figură este arborele asociat prin strategie șirului de vectori  $x(i1), \dots, x(in)$ .



Arborele construit pe baza a doi vectori



Arborele construit pe baza a două subsecvențe de vectori.

Arborele asociat unui șir de vectori verifică:

- are atâtea frunze câți vectori sunt în șirul de vectori;
- informația atașată unui nod este suma dimensiunilor vectorilor din subarborii stâng și drept ai nodului;
- dacă un vector apare pe nivelul  $p$  atunci termenii șirului apar în exact  $p$  interclasări;
- numărul total de comparații 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 de Corectitudine.** Algoritmul InterOpt determină interclasarea vectorilor  $x(i)$ ,  $i = 1, n$  cu număr minim de comparații realizate din interclasările directe.

*Demonstrație.* Se observă că arborele asociat unei strategii de interclasare este arborele Huffman asociat numerelor  $nr(1), \dots, nr(n)$ . Algoritmul anterior repetă modul de construcție al arborelui Huffman deci suma  $\sum_{i=1}^n nr(i) \cdot niv(i)$  este minimă și implicit  $L(T)$  este minim.

Rezultă că strategia de interclasare este optimă.

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

*Demonstrație.* Complexitatea algoritmului este dată de numărul de comparații făcut de interclasările directe și cele date de păstrarea structurii  $x$ . În primul caz numărul este  $n$ . Pentru cel de al doilea caz se observă că inserarea directă folosește cel mult  $i-1$  comparații pentru punerea șirului  $y$  în structura  $x$ . Găsim un număr total de  $\sum_{i=2}^n (i-1) = \Theta(n^2)$  comparații.

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



### Rezumat

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





### M2.U5.3. Test de autoevaluare a cunoștințelor

1. Dacă  $x = (x(i), i = 1, m)$ ,  $y = (y(i), i = 1, n)$  reprezintă două mulțimi de elemente, atunci să se determine mulțimea intersecție  $z = (z(i), i = 1, k)$ .
2. Dându-se  $n$  obiecte de cost  $c = (c(i), i = 1, n)$  cu greutatea  $g = (g(i), i = 1, n)$  și 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 cunoaște intervalul în care se desfășoară:  $[st, sf)$ . Se cere să se planifice un număr maxim de spectacole astfel încât să nu se suprapună..
4. Se dau  $n$  numere întregi nenule  $b_1, \dots, b_n$  și  $m$  numere întregi nenule  $a_1, \dots, a_m$ ,  $n \geq m$ . Să se determine o submulțime a mulțimii  $B = \{b_1, \dots, b_n\}$  care să maximizeze valoarea expresiei:  $E = a_1 * x_1 + a_2 * x_2 + \dots + a_m * x_m$  unde  $x_i \in B$ .
5. O stație de servire trebuie să satisfacă cererile a  $n$  clienți. Timpul de servire pentru fiecare client este cunoscut: pentru clientul  $i$  timpul este  $t_i$ . Să se minimizeze timpul total de așteptare.  $T = \sum_{i=1}^n$  (timpul de așteptare pentru clientul  $i$ )

## ***M2.U6. Metoda programării dinamice.***



### **M2.U6.1. Introducere**

Programarea dinamică, asemenea metodelor “*Divide et Impera*”, rezolvă problemele combinând soluțiile unor subprobleme. “Programarea” în acest context se referă la o metodă tabelară și nu la scrierea unui cod pentru calculator.



### **M2.U6.2. Obiectivele unității de învățare**

La sfârșitul acestei unități de învățare studenții vor fi capabili să:

- fie familiarizat cu trăsăturile generale ale unei probleme pentru care se apelează la programare dinamică
- își însușească strategia generală de rezolvare a unei probleme folosind această metodă.



**Durata medie de parcurgere a acestei unități de învățare este de trei ore.**

Am văzut, în capitolul referitor la *divide et impera*, că algoritmi construiți cu această metodă partiționează problema în subprobleme independente pe care le rezolvă în general recursiv, după care combină soluțiile lor pentru a obține soluția problemei inițiale. Spre deosebire de această abordare, programarea dinamică este aplicabilă atunci când aceste subprobleme nu sunt independente, adică au în comun subproblemele.

În general, programarea dinamică se aplică problemelor de optimizare. Fiecare soluție are o valoare și se dorește determinarea soluției optime (minim sau maxim). O asemenea soluție numește soluție optimă a problemei, prin contrast cu valoarea optimă, deoarece pot exista mai multe soluții care să realizeze valoarea optimă.

Dezvoltarea unui algoritm bazat pe metoda programării dinamice poate fi împărțită într-o secvență de patru pași:

1. Caracterizarea structurii unei soluții optime
2. Definirea recursivă a valorii unei soluții optime
3. Calculul valorii unei soluții optime într-o manieră “bottom-up” (plecând de la subprobleme de dimensiune mică și ajungând la unele de dimensiuni din ce în ce mai mari)
4. Construirea unei soluții optime din informația calculată.

Pașii 1–3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se dorește doar calculul valorii optime. În vederea realizării pasului 4, deseori se păstrează

informație suplimentară de la execuția pasului 3, pentru a ușura construcția unei soluții optime.

### Înmulțirea unui șir de matrici

Primul nostru exemplu de programare dinamică este un algoritm care rezolvă problema înmulțirii unui șir de matrici. Se dă un șir  $A_1, A_2, \dots, A_n$  de  $n$  matrici care trebuie înmulțite. Acest produs se poate evalua folosind algoritmul clasic de înmulțire a unei perechi de matrici ca subalgoritm, o dată ce produsul  $A_1 \cdot \dots \cdot A_n$  este parantezat (această restricție este impusă de faptul că înmulțirea matricilor este operație binară). Un produs de matrici este complet parantezat dacă este format dintr-o singură matrice sau dacă este format din factori compleți parantezați.



Exemplu.

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

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

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

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

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

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



Aplicați exemplul de mai sus pentru 4 matrici.

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

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

dacă  $m \neq p$  atunci

    scrie( "dimensiuni incompatibile" )

altfel

pentru  $i = 1, n$

pentru  $j = 1, q$

$C(i, j) = 0$

pentru  $k = 1, m$

$C(i, j) = C(i, j) + A(i, k) \cdot B(k, j)$

sfârșit pentru

sfârșit pentru

sfârșit dacă

### Return

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



### Exemplu

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



Să considerăm problema șirului  $A_1, A_2, A_3, A_4$ . Să presupunem că dimensiunile matricilor sunt  $10 \times 100$ ,  $100 \times 5$ ,  $5 \times 50$ ,  $50 \times 2$ . Efectuați calculele ca mai sus.

Enunțul formalizat al problemei este: dându-se un șir  $A_1, A_2, \dots, A_n$  de  $n$  matrici, unde pentru  $i = 1, n$  matricea  $A_i$  are dimensiunile  $p_{i-1} \times p_i$ , să parantezeze complet produsul  $A_1 \cdot A_2 \cdot \dots \cdot A_n$  astfel încât să se minimizeze numărul de înmulțiri scalare. Înainte de a rezolva problema înmulțirii șirului 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 secvențe de  $n$  matrice. O secvență de  $n$  matrice o putem diviza între matricele  $k$  și  $k + 1$  pentru orice  $k = 1, \dots, n - 1$  și apoi putem descompune în paranteze, în mod independent, fiecare dintre secvențe.

În acest fel vom obține următoarea recurență:

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

Numerele astfel definite se numesc *numerele lui Catalan* și se arată că  $P(n) = C(n-1)$  unde

$$C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega(4^n / n^{3/2})$$

Numărul de soluții este exponențial, deci ar fi o strategie slabă. Primul pas în schema generală a metodei programării dinamice este dat de caracterizarea unei soluții optimale. Pentru problema noastră descrierea este făcută în continuare. Vom face următoarea convenție

de notare: matricea obținută în urma procesului de evaluare a produsului  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$  se va nota cu  $A_{i..j}$ . O parantezare optimă a produsului  $A_1 \cdot \dots \cdot A_n$  împarte produsul între  $A_k$  și  $A_{k+1}$  pentru un anumit  $k$  din intervalul  $1 \dots n$ . Aceasta înseamnă că pentru o valoare a lui  $k$ , mai întâi calculăm matricele  $A_{1..k}$  și  $A_{k+1..n}$  și apoi le înmulțim pentru a obține rezultatul final  $A_{1..n}$ . Costul acestei parantezări optime este dat de suma costurilor calculului pentru matricele  $A_{1..k}$ ,  $A_{k+1..n}$  și  $A_{1..n}$ . Observația care trebuie făcută este că: parantezarea subșirului prefix  $A_1 \cdot \dots \cdot A_k$  în cadrul parantezării optime a produsului  $A_1 \cdot \dots \cdot A_n$  este o parantezare optimă pentru  $A_1 \cdot \dots \cdot A_k$ , pentru că altfel, dacă am presupune că mai există o metodă de parantezare mai puțin costisitoare a lui  $A_1 \cdot \dots \cdot A_k$  ar contrazice faptul că parantezarea pentru  $A_1 \cdot \dots \cdot A_n$  este optimă. Această observație este valabilă și pentru parantezarea lui  $A_{k+1} \cdot \dots \cdot A_n$ . Prin urmare, o soluție optimă a unei instanțe a unei probleme conține soluții optime pentru instanțe ale subproblemelor. Al doilea pas în aplicarea metodei programării dinamice este dat de definirea valorii unei soluții optime în mod recursiv, în funcție de soluțiile optime ale subproblemelor. În cazul problemei înmulțirii șirului de matrice, o subproblemă constă în determinarea costului minim al unei parantezări a șirului  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ , pentru  $1 \leq i \leq j \leq n$ . Fie  $m(i, j)$  numărul minim de înmulțiri scalare necesare pentru a calcula matricea  $A_{i..j}$ ; costul modalităților optime de calcul al lui  $A_{1..n}$  va fi  $m(1, n)$ . Definiția recursivă a lui  $m(i, j)$  este dată de formula

$$m[i, j] = \begin{cases} 0 & \text{daca } i = j \\ \min \{ m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \} & \text{daca } i < j \end{cases}$$

Valorile  $m(i, j)$  exprimă costul soluțiilor optime ale subproblemelor. Pentru a putea urmări modul de construcție a soluției optime, să definim  $s(i, j)$  care va conține valoarea  $k$  pentru care împărțirea produsului  $A_i \cdot \dots \cdot A_j$  produce o parantezare optimă. Aceasta înseamnă că  $s(i, j)$  este egal cu valoarea lui  $k$  pentru care  $m(i, j) = m(i, k) + m(k+1, j) + p_{i-1} p_k p_j$ .

În acest moment este ușor să scriem un algoritm recursiv dat de recurența de mai sus care să calculeze costul minim  $m(1, n)$  pentru produsul  $A_1 \cdot \dots \cdot A_n$ . Dar acest algoritm necesită timp exponențial - nu mai bun decât căutarea completă a parantezărilor. Observația care se impune în acest moment se referă la faptul că avem relativ puține subprobleme: o problemă

pentru fiecare alegere a lui  $i$  și  $j$  ce satisfac  $1 \leq i \leq j \leq n$ , adică un total de  $\binom{n}{2} + n = \Theta(n^2)$ .

Un algoritm recursiv poate întâlni fiecare subproblemă de mai multe ori pe ramuri diferite ale arborelui său de recurență. Această proprietate de suprapunere a subproblemelor este a doua caracteristică a programării dinamice. În loc să calculăm recursiv soluția recurenței vom aplica pasul al treilea din schema programării dinamice și 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  $i = 1, \dots, n$ . Intrarea este secvența  $(p_0, \dots, p_n)$  de  $n+1$  elemente. Procedura folosește un tablou auxiliar  $m(1 \dots n, 1 \dots n)$  pentru costurile  $m(i, j)$  și un tablou auxiliar  $s(1 \dots n, 1 \dots n)$  care înregistrează acea valoare a lui  $k$  pentru care s-a obținut costul optim în calculul lui  $m(i, j)$

InmultireMatrici( $p, m, s, n$ )

```

pentru  $i = 1, n$ 
     $m(i, i) = 0$ 
sfârșit pentru
pentru  $l = 2, n$ 
    pentru  $i = 1, n - l$ 
         $j = i + l - 1$ 
         $m(i, j) = \infty$ 
        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
                     $m(i, j) = q$ 
                     $s(i, j) = k$ 
            sfârșit dacă
        sfârșit pentru
    sfârșit pentru
Return

```

Algoritmul completează tabloul  $m$  într-un mod ce corespunde rezolvării problemei parantezării unor șiruri de matrici din ce în ce mai mari. Relația arată că  $m(i, j)$ , costul de calcul al sirului de  $j - i + 1$  matrici depinde doar de costurile calculării produselor șirurilor de mai puțin de  $j - i + 1$  matrici. Aceasta înseamnă că, pentru  $k = i, i + 1, \dots, j - 1$ , matricea  $A_i \dots k$  este un produs de  $k - i + 1 < j - i + 1$  matrice, iar matricea  $A_{k+1} \dots j$  este un produs de  $j - k < j - i + 1$  matrici. În ciclul *pentru*  $i = 1, n$  algoritmul inițializează  $m(i, i) = 0$  (costul minim al șirurilor de lungime 1). La prima execuție a ciclului *pentru*  $l = 2, n$  se calculează, cu formula  $m(i, i + 1)$  pentru  $i = 1, 2, \dots, n - 1$  (costul minim al șirurilor de lungime 2). La a doua trecere prin ciclul *pentru*  $l = 2, n$  se calculează  $m(i, i + 2)$ , pentru  $i = 1, 2, \dots, n - 2$  (costul minim al șirurilor de lungime 3), etc. La fiecare pas, costul  $m(i, j)$  calculat în ciclul *pentru*  $k = i, j - 1$  depinde doar de intrările  $m(i, k)$  și  $m(k + 1, j)$  ale tabloului, deja calculate.



### Exemplu

În tabelul de mai jos este descrisă funcționarea algoritmului pentru un șir de  $n = 6$  matrici având dimensiunile date mai jos.

Matrice Dimensiune

$A_1$	$30 \times 35$
$A_2$	$35 \times 15$
$A_3$	$15 \times 5$
$A_4$	$5 \times 10$
$A_5$	$10 \times 20$
$A_6$	$20 \times 25$

Tabelul cu Dimensiunile matricelor care se înmulțesc



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	1050	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

Evoluția algoritmului pentru matricile de mai sus. În partea stângă apare matricea  $m$ , în partea dreaptă matricea  $s$ . În tablourile  $m, s$  sunt utilizate doar elementele de deasupra diagonalei principale. Numărul minim de înmulțiri scalare necesare înmulțirii acestor șase 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:

$$m(2, 2) + m(3, 5) + p_1p_2p_5 = 13000$$

$$m(2, 3) + m(4, 5) + p_1p_3p_5 = 7125$$

$$m(2, 4) + m(5, 5) + p_1p_4p_5 = 11375$$

$$\text{rezultă } m(2,5) = 7125$$

O simplă examinare a algoritmului conduce la constatarea că timpul de execuție este  $\Theta(n^3)$ . Deși acest algoritm determină numărul optim de înmulțiri scalare necesare pentru calculul produsului șirului de matrici, acesta nu prezintă în mod direct modul în care se face înmulțirea. Pasul 4 al schemei generale a metodei programării dinamice urmărește construirea unei soluții optime din informația disponibilă. În acest caz particular, vom folosi tabloul  $s$  pentru a determina modul optim de înmulțire a matricilor. Fiecare element  $s(i, j)$  conține valoarea lui  $k$  pentru care parantezarea optimă a produsului  $A_i \cdot \dots \cdot A_j$  împarte produsul între  $A_k$  și  $A_{k+1}$ . Atunci știm că în produsul final de calcul al matricel  $A_1 \dots A_n$  optimul este  $A_1 \dots s(1, n) \cdot A_{s(1, n)+1} \dots A_n$ . Înmulțirile anterioare pot fi determinate recursiv, deoarece  $s(1, s(1, n))$  determină ultima înmulțire matriceală din calculul lui  $A_1 \dots s(1, n)$ ; analog,  $s(s(1, n)+1, n)$  determină ultima înmulțire din produsul  $A_{s(1, n)+1} \dots A_n$ .

### Cel mai lung subșir crescător

Problema se enunță în felul următor: se dă un șir de numere  $a = (a(i), i = 1, n)$ . Să se determine cel mai lung subșir crescător. Prin subșir se înțelege un șir  $a(i_1), a(i_2), \dots, a(i_k)$ , unde  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  (elementele subșirului nu sunt neapărat adiacente în șirul inițial). Dacă am considera toate subșirurile de elemente, atunci acestea ar fi în număr de  $2^n - 1$ . Chiar și dacă am genera prin backtracking subșirurile crescătoare ale lui  $a$ , tot la complexitate exponențială s-ar ajunge. Ideea este de a calcula, pentru fiecare element al șirului  $a$  cel mai lung subșir crescător care începe cu elementul respectiv. În final, se alege cel mai mare subșir

crescător din cele determinate pentru fiecare element. Să presupunem că cel mai lung subșir crescător care conține elementul  $a(p)$  este  $a(i_1), \dots, a(p), \dots, a(i_k)$  (1). Facem afirmația că subșirul  $a(p), \dots, a(i_k)$  este cel mai lung subșir crescător care se formează începând cu  $a(p)$ . Dacă prin absurd nu ar fi așa, atunci ar însemna că am avea un alt subșir crescător care să înceapă cu  $a(p)$ :  $a(p), \dots, a(i_k)$  (2). Ori, dacă am considera subșirul  $a(i_1), \dots, a(p), \dots, a(i_k)$  (partea finală a lui (1) este înlocuită cu subșirul (2)), atunci am avea un subșir crescător care să îl conțină 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. Afirmațiile de mai sus reprezintă suportul pentru strategia de rezolvare aleasă.

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

$$L(i) = \begin{cases} 1 & \text{daca } i = n \\ 1 + \max\{L(j) \mid j > i, a(j) > a(i), \text{daca } i < n\} \end{cases}$$

unde pentru cazul mulțimii 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 aceleași valori), în ordinea  $L(n), L(n-1), \dots, L(1)$ . Pentru a determina mai ușor subșirul de lungime maximă, se construiește un șir  $sol = (sol(i), i = 1, n)$  care va conține pentru valoarea  $i$  acel  $j$  care dă minimul pentru (2.6.1) sau chiar valoarea  $i$  dacă un asemenea  $j$  nu există. Determinarea lungimii maxime a unui subșir se face apoi calculând  $\max_{i=1,n}(L(i))$  iar determinarea efectivă a acestui șir se face pe baza informației din  $sol$ .



### Exemplu

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

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

Valoarea maximă din vectorul  $L$  este 3, obținută pentru  $L(1)$  (și altele). Pe baza vectorului  $sol$  obținem și șirul 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.

Aplicați algoritmul pentru șirul  $a = (5, 3, 6, 1, 3, 4, 7)$





Exemplu:

*Start SubsirCrescator*

citește(  $n$ , ( $a(i)$ ,  $i = 1, n$ ) )

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  $a(j) \geq a(i)$  și  $1 + L(j) > L(i)$  atunci

$L(i) = 1 + L(j)$

$sol(i) = j$

sfârșit dacă

sfârșit pentru

sfârșit pentru

$LMax = L(1)$

$pozLMax = 1$

pentru  $i = 2, n$

dacă  $L(i) > LMax$  atunci

$LMax = L(i)$

$pozLMax = i$

sfârșit dacă

sfârșit pentru

scrie( “Lungimea maximă a unui subșir crescător este:”,  $LMax$  )

scrie(  $a(LMax)$  )

$i = pozLMax$

cât timp  $sol(i) \neq i$

$i = sol(i)$

scrie(  $a(i)$  )

sfârșit cât timp

Stop

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



## Rezumat

Am văzut, în capitolul referitor la *divide et impera*, că algoritmii construiți cu această metodă partiționează problema în subprobleme independente pe care le rezolvă în general recursiv, după care combină soluțiile lor pentru a obține soluția problemei inițiale. Spre deosebire de această abordare, programarea dinamică este aplicabilă atunci când aceste subprobleme nu sunt independente,

adică au în comun subprobleme.

În general, programarea dinamică se aplică problemelor de optimizare.

Fiecare soluție are o valoare și se dorește determinarea soluției optime (minim sau maxim). O asemenea soluție numește soluție optimă a problemei, prin contrast cu valoarea optimă, deoarece pot exista mai multe soluții care să realizeze valoarea optimă.

Programarea dinamică este o metodă care se folosește în cazurile în care optimul general implică optimul parțial; în astfel de cazuri, demonstrația se face de cele mai multe ori prin reducere la absurd. Pentru evitarea calculării acelorași rezultate, se apelează la tabelare – rezultatele pentru cazurile luate în considerare sunt memorate într-o structură de tip matricial. Pe baza acestor informații, se poate determina soluția (soluțiile, chiar) pentru care se atinge optimul.



### **M2.U6.3. Test de autoevaluare a cunoștințelor**

1. Găsiți o parantezare optimă a produsului unui șir de matrice al cărui șir de dimensiuni este (5, 10, 3, 12, 5, 50, 6) (adică  $5 \times 10$ ,  $10 \times 3$ , etc).
2. Arătați că o parantezare completă a unei expresii cu  $n$  elemente are exact  $n - 1$  perechi de paranteze.
3. 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-șila-dreapta. Se cere un algoritm care găsește suma maximă.

Exemplu:

5

2 3

8 1 2

5 9 3 4

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

4. Folosind rezultatele lui InmultireMatrici să se descrie un algoritm care realizează înmulțirea matricilor în ordine optimă.

## Răspunsuri la teste



### ***M1.U2.3. Test de autoevaluare a cunoștințelor***

1) Se definește produsul scalar a doi vectori A și B, de aceeași dimensiune n cu formula:

$$PS = \sum_{i=1}^n A(i) \cdot B(i).$$

Să se scrie algoritmul pentru calculul produsului scalar.

- 2). Se dau n, vectorul A(i), i=1, n și un număr x. Să se scrie un algoritm care găsește indicele j pentru care A(j)=x (problema căutării).
3. Cum se modifică algoritmul de mai sus dacă știm că vectorul este sortat crescător?
4. Se dau n și vectorul A(i), i=1, n. Să se găsească cel mai mic element.
5. Reluați problema anterioară în situația în care trebuie să aflați câte elemente minime sunt. Încercați să rezolvați problema folosind o singură parcurgere a șirului.
6. Se dau n și vectorul A(i), i=1, n. Ce număr apare cel mai des și de câte ori?
7. Cum se modifică algoritmul precedent dacă vectorul este deja sortat ?

### **Răspunsuri:**

#### **1. Start ProdusScalar**

Citește (n, (a(i), i = 1, n), (b(i), i = 1, n) )

ps = 0 {0 este element neutru pentru adunare}

pentru i = 1, n

ps = ps + a(i) · b(i)

sfârșit pentru

Scrie ('Produsul scalar este: ', ps )

Stop

Complexitatea este  $\Theta(n)$ .

2. Ideea este de a parcurge șirul și a compara fiecare element cu x; dacă acest element este găsit, atunci este oprită căutarea, altfel se merge la următorul element. La sfârșit se raportează rezultatul. Pentru a semnaliza faptul că x a fost găsit, vom folosi o variabilă booleană găsit care va primi valoarea adevărat dacă x a fost găsit în șir, fals în caz contrar.

#### **Start CăutareLiniară**

Citește (n, (a(i), i = 1, n), x )

găsit = fals {deocamdată nu am găsit pe x în a}

i = 1 {i este indicele elementului curent din șir}

repetă

dacă a(i) = x atunci

găsit = adevărat

poziție = i

altfel  
 $i = i + 1$   
sfârșit dacă  
până când ( $i > n$ ) sau (găsit = adevărat)  
dacă găsit = adevărat atunci  
Scrie (‘Am găsit’, x, ‘pe poziția’, poziție)  
altfel  
Scrie (x, ‘nu a fost găsit’)  
sfârșit dacă  
Stop  
 Complexitatea este  $\Theta(n)$ , cazul cel mai defavorabil fiind când x nu se află în șir.  
*Observație:* Deoarece execuția instrucțiunilor din interiorul ciclului trebuie să se facă cel puțin o dată, este corectă utilizarea unui ciclu repetă.

**3.** Vom folosi un algoritm asemănător cu cel de la problema precedentă, folosind informația că șirul este ordonat crescător. Vom da două variante, cu analiză de complexitate.

**(a) Prima variantă:** începem căutarea, dar vom impune o condiție 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  
Citește (n, (a(i),  $i = 1, n$ ), x)  
 găsit = fals {deocamdată nu am găsit pe x în a}  
 $i = 1$  {i este indicele elementului curent din șir}  
repetă  
dacă  $a(i) = x$  atunci  
     găsit = adevărat  
     poziție = i  
altfel  
      $i = i + 1$   
sfârșit dacă  
până când ( $i > n$ ) sau (găsit = adevărat) sau ( $x > a(i)$ )  
dacă găsit = adevărat atunci  
Scrie (‘Am găsit’, x, ‘pe poziția’, poziție)  
altfel  
Scrie (x, ‘nu a fost găsit’)  
sfârșit dacă  
Stop  
 Complexitatea este aceeași ca la algoritmul precedent:  $\Theta(n)$ .  
*Observație:* Este esențial 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 condiția  $a(i) \leq x$  să fie testată, deși  $a(n + 1)$  nu există.

**(b) A doua variantă:** speculează mai inteligent faptul că șirul este gata sortat. Putem împărți șirul în două subșiruri de dimensiuni cât mai apropiate. Se testează dacă elementul de la mijloc este egal cu x. Dacă da, atunci căutarea se oprește cu succes. Dacă nu, și  $x <$  elementul din mijloc, atunci căutarea va continua să se facă în prima jumătate a șirului (e clar că în a doua jumătate nu poate fi găsit); dacă x este mai mare decât elementul de la jumătate, atunci căutarea va continua în a doua jumătate. Căutarea continuă până când fie se găsește x în șir, fie nu mai avem nici un element de cercetat, caz care înseamnă căutare fără succes. Acest algoritm se numește algoritmul căutării prin înjumătățire.

Start CăutareÎnjumătățire

Citește (n, (a(i), i = 1, n), x )

găsit = fals { ca în algoritmul anterior }

stânga = 1

dreapta = n {stânga și dreapta reprezintă capetele între care se face căutarea lui x}

{Inițial, căutarea se face în tot vectorul}

cât timp (găsit = fals) și (stânga ≤ dreapta)

{nu l-am găsit pe x și mai avem unde căuta}

mijloc = \_stânga+dreapta

{mă poziționez la jumătatea șirului}

dacă a(mijloc) = x atunci

găsit = adevărat

altfel

dacă x < a(mijloc) atunci

dreapta = mijloc - 1

altfel

stânga = mijloc + 1

sfârșit dacă

{caut doar în jumătatea corespunzătoare}

sfârșit dacă

sfârșit cât timp

dacă găsit = adevărat atunci

Scrie ( ' Am găsit ' , x, ' pe poziția ' , poziție )

altfel

Scrie (x, ' nu a fost găsit ' )

sfârșit dacă

Stop

Complexitate: Timpul T(n) pentru rezolvarea problemei de dimensiune n satisface ecuația:  $T(n) = T(n/2) + c$ , c fiind o constantă necesară testărilor și atribuirilor (operații elementare, al căror cost în parte nu depinde de n). Prin inducție completă se poate demonstra că:

$T(n) = \Theta(\log_2(n))$ , adică mult mai performant decât în varianta 3.

4. Presupunem că minimul este primul element din șir. Luăm apoi fiecare din celelalte elemente ale vectorului  $a$  și 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. Procedul continuă până când se epuizează elementele din șir. La sfârșit minimul găsit va fi de fapt minimul absolut al șirului, ceea ce trebuia determinat.

Start Minim

Citește ( $n, (a(i), i = 1, n)$  )

minim =  $a(1)$

pentru  $i=2, n$

dacă  $a(i) < \text{minim}$  atunci

minim =  $a(i)$

sfârșit dacă

sfârșit pentru

Scrie (‘minimul este ’, minim )

Stop

Numărul de comparații este  $n - 1$ . Nu se poate coborî sub acesta (demonstrația se face prin inducție).

*Observație.* Există varianta de a inițializa minim cu o valoare foarte mare, considerată  $+\infty$  pentru tipul de date care este folosit. De multe ori limbajele de programare pun la dispoziție constante reprezentând valorile cele mai mari (mici) reprezentabile pe un anumit tip de date numeric (Pascal, C/C++, Java).

5. Vom folosi un contor (o variabilă) pentru a urmări de câte ori apare minimul.

(a) Vom rezolva problema în doi pași: vom determina minimul printr-o parcurgere a șirului, ca mai sus, iar apoi vom parcurge șirul încă o dată pentru a număra de câte ori a apărut acesta.

Start ContorMinim1

Citește ( $n, (a(i), i = 1, n)$  )

minim =  $a(1)$

pentru  $i=2, n$

dacă  $a(i) < \text{minim}$  atunci

minim =  $a(i)$

sfârșit dacă

sfârșit pentru

contor = 0

pentru  $i = 2, n$

dacă minim =  $a(i)$  atunci

contor = contor + 1

sfârșit dacă

sfârșit pentru

Scrie (‘minimul este ’, minim )

Scrie (‘apare de ’, contor, ’ ori’ )

Stop

Complexitate:  $\Theta(n)$ , deoarece șirul se parcurge de două ori, pentru fiecare element efectuându-se operații elementare ce nu depind de  $n$  (dimensiunea intrării).

(b) Vom rezolva problema printr-o singură parcurgere a șirului; complexitatea teoretică rămâne aceeași.

Start ContorMinim2

Citește ( $n, (a(i), i = 1, n)$  )

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}

sfârșit dacă

sfârșit dacă

sfârșit pentru

Scrie (‘minimul este ’, minim )

Scrie (‘apare de ’, contor, ’ ori’ )

Stop

6. Vom construi doi vectori: unul reține numerele distincte din  $a$ , iar celălalt contorizează numărul de apariții pentru fiecare număr. Vom scrie un subalgoritm de tip funcție de căutare liniară a unui element  $x$  într-un vector  $v$  care conține un anumit număr de elemente. Această funcție returnează poziția pe care se găsește elementul  $x$  în  $v$  sau 0 în cazul în care  $x$  nu este cuprins în  $v$ :

CautăLiniar ( $x, v, l$ )

poziție = 0

găsit = fals

$i = 1$

repetă

dacă  $x = v(i)$  atunci

găsit = adevărat

poziție =  $i$

altfel

$i = i + 1$

sfârșit dacă

până când ( $i > l$ ) sau (găsit = adevărat)

CautăLiniar = poziție

### Return

### Start CelMaiDes

Citește (n, (a(i), i = 1, n) )

k = 0

{k este numărul de elemente distincte din a}

pentru i = 1, n

PozInDif = CautăLiniar (a(i), diferite, k)

dacă PozInDif = 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 dată înainte în a}

contor(PozInDif) = contor(PozInDif)+1

{l-am mai găsit o dată}

sfârșit dacă

sfârșit pentru

{acum căutăm în vectorul contor să vedem care număr a apărut cel mai des}

maximApariții = contor(1)

pozMax = 1

pentru i = 2, k

dacă contor(i) > maximApariții atunci

maximApariții = contor(i)

pozMax = i

sfârșit dacă

sfârșit pentru

Scrie ('Numărul cel mai frecvent:', diferite(pozMax))

Scrie ('Apare de ', maximApariții, ' ori' )

Stop

Complexitate: cazul cel mai defavorabil este atunci când avem doar elemente distincte în șirul a. În acest caz, un apel CautăLiniar(a(i), diferite, k) are costul de forma  $a \cdot k + b$ , a și 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

frecvența: 1

7. Vom profita de faptul că vectorul dat este sortat crescător. Vom căuta o secvență de numere constante, de lungime maximă.

Start CelMaiDes

StartSir = 1

EndSir = 1

lungimeSecventaMaxima = 0

suntInSir = adevărat

cât timp suntInSir = adevărat

cât timp (EndSir)  $\leq$  n și (a(StartSir) = a(EndSir))

EndSir = EndSir + 1

sfârșit cât timp

dacă EndSir – StartSir > LungSecvMax atunci

apareMaxim = a(StartSir)

LungSecvMax = EndSir – StartSir

sfârșit dacă

StartSir = EndSir

dacă StartSir > n atunci

suntInSir = fals

{am parcurs tot șirul} sfârșit dacă

sfârșit cât timp

Scrie ('Numărul cel mai frecvent:', apareMaxim)

Scrie ('Apare de', LungSecvMax, ' ori' )

Stop

Complexitate: se observă că valoarea EndSir ia pe rând valorile 1, 2, . . . , n+1; de fiecare dată sunt efectuate operații elementare, deci avem  $T(n) = \Theta(n + 1) = \Theta(n)$ .

*Observație:* Pentru a rezolva problema de la punctul anterior, am putea face o sortare a șirului, 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 obținut anterior.



### ***M1.U4.3. Test de autoevaluare a cunoștințelor***

1. Scrieți algoritmi de intrare/ieșire din coada circulară.
2. Scrieți algoritmi iterativi pentru parcurgerea arborilor binari în inordine și postordine.
3. Se dă o stivă alocată în lăntuit și o valoare x. Să se extragă din stivă, dacă există, elementul cu INFO = x.

4. Se dă o stivă și 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.

#### **Răspunsuri:**

1. Vom folosi pentru memorarea elementelor din coadă un vector  $v$  de  $n$  elemente. Vom presupune că indicii vectorului sunt între 0 și  $n - 1$ . Deoarece coada este circulară, după locația de indice  $n - 1$  urmează locația de indice 0. Inițial, coada este vidă, fapt semnalat prin  $inceput = sfarsit = 0$ .

$x \Rightarrow Coadă$

$sfarsit = (sfarsit + 1) \bmod n$

dacă  $sfarsit = inceput$  atunci

“Coadă este plină”

altfel

$v(sfarsit) = x$

sfârșit dacă

Return

$x \Leftarrow Coadă$

dacă  $inceput = sfarsit$  atunci

“Coadă este vidă”

altfel

$inceput = (inceput + 1) \bmod n$

$x = v(inceput)$

sfârșit dacă

Return

Complexitatea fiecărui subalgoritm este  $\Theta(1)$ . Inițial avem  $inceput = sfarsit = 0$ .

*Observație:* Se folosesc efectiv doar  $n - 1$  locații ale vectorului; dar în acest mod se poate distinge între situațiile de coadă plină și coadă vidă.

#### **2. (a) Parcurgerea în inordine, iterativ:**

$InordineIterativ(rad)$

$i = rad$

$S = \emptyset$  {  $S$  este o stivă }

repetă

cât timp  $LS(i) \neq *$

$i \Rightarrow S$

$i = LS(i)$

sfârșit cât timp

scrie ( $INFO(i)$ )

cât timp  $LD(i) = *$

dacă  $S = \emptyset$  atunci  
Return{ se iese din subalgoritm, deci și din ciclul infinit}  
sfârșit dacă  
 $i \Leftarrow S$   
scrie( INFO( $i$ ) )  
sfârșit cât timp  
 $i = LD(i)$   
până când fals{ciclul 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  $LS(i) \neq *$   
 $i \Rightarrow S$   
 $i = LS(i)$   
sfârșit cât timp  
cât timp  $LD(i) = *$   
repetă  
scrie( INFO( $i$ ) )  
dacă  $S = \emptyset$  atunci  
Return{se iese din subalgoritm, deci și din ciclul infinit}  
sfârșit dacă  
 $j = i$   
 $i \Leftarrow S$   
până când  $j = LS(i)$   
sfârșit cât timp  
 $i \Rightarrow S$   
 $i = LD(i)$   
până când fals{ciclul 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.** Vom extrage succesiv vârful stivei  $S$  și 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  $S\_ = \emptyset$  și  $gasitx = fals$

$a \leftarrow S$  { scoatem vârful stivei }

dacă  $a\_ = x$  atunci

$a \Rightarrow T$  { îl depunem în  $T$  }

altfel

$gasitx = adevarat$

sfârșit dacă

sfârșit cât timp

cât timp  $T\_ = \emptyset$

$a \leftarrow T$

$a \Rightarrow S$

sfârșit cât timp

Return

Complexitate: cazul cel mai defavorabil este acela în care  $x$  nu se găsește în  $S$ . În acest caz, tot conținutul lui  $S$  este trecut în  $T$  și apoi repus în  $S$ . Complexitatea este deci  $\Theta(|S|)$ , unde  $|S|$  reprezintă numărul de elemente aflate inițial în  $S$ .

*Observație:* Primul ciclu *cât timp* poate fi scris mai adecvat ca un ciclu *repetă*, având în vedere faptul că execuția ciclului se face cel puțin o dată.

4. Rezolvarea este asemănătoare cu cea de la punctul precedent.

Insereaza(  $S, x, y$  )

$T = \emptyset$

$gasitx = fals$

cât timp  $S\_ = \emptyset$  și  $gasitx = fals$

$a \leftarrow S$  { scoatem vârful stivei }

dacă  $a\_ = x$  atunci

$a \Rightarrow T$  { îl depunem în  $T$  }

altfel

$gasitx = adevarat$

$a \Rightarrow S$

sfârșit dacă

sfârșit cât timp

dacă  $gasitx = adevarat$  atunci

$y \Rightarrow S$

sfârșit dacă

cât timp  $T\_ = \emptyset$

$a \leftarrow T$

$a \Rightarrow S$

sfârșit cât timp

Return

Complexitatea este  $\Theta(|S| + 1) = \Theta(|S|)$ , motivația fiind asemănătoare cu cea de la punctul precedent.

**5.** Vom folosi două stive: *Stiva1*, *Stiva2*.

$x \Rightarrow Coda$

dacă *Stiva1* =  $\emptyset$  atunci

Golește(*Stiva2*, *Stiva1*) {Golește elementele din *Stiva2* în *Stiva1*}

sfârșit dacă

$x \Rightarrow Stiva1$

Return

$x \Leftarrow Coda$

dacă *Stiva2* =  $\emptyset$  atunci

Golește(*Stiva1*, *Stiva2*) {Golește elementele din *Stiva1* în *Stiva2*}

sfârșit dacă

dacă *Stiva2* =  $\emptyset$  atunci

“Coda este vida”

altfel

$x \Leftarrow Stiva2$

sfârșit dacă

Return

Subalgoritmul Golește(*A*,*B*), unde *A* și *B* sunt stive, va trece toate elementele din *A* în *B*, prin extrageri repetate.

Golește(*A*,*B*)

cât timp *A*  $\neq \emptyset$

$x \Leftarrow A$

$x \Rightarrow B$

sfârșit cât timp

Return

Numărul de operații de inserare și extragere în fiecare caz zunt:

(a) Subalgoritmul Golește(*A*,*B*) efectuează  $2|A|$  extrageri ( $|A|$  este numărul de elemente din stiva *A*);

(b) Subalgoritmul  $x \Rightarrow Coda$  efectuează  $2|Stiva2|+1=2|Coda|+1$  operații de inserare/extragere;

(c) Subalgoritmul  $x \Leftarrow Coda$  efectuează  $2|Stiva1|+1=2|Coda|+1$  operații de inserare/extragere;

**6.** Vom folosi două cozi: *Coda1*, *Coda2*.

$x \Rightarrow Stiva$

$x \Rightarrow Coadal$

*Return*

$x \Leftarrow Stiva$

dacă  $Coadal = \emptyset$  atunci

“eroare: stiva vidă”

altfel

cât timp  $Coadal \neq \emptyset$

$x \Leftarrow Coadal$

dacă  $Coadal \neq \emptyset$  atunci  $\{x$  nu este ultimul element din  $Coadal\}$

$x \Rightarrow Coadal2$

sfârșit dacă

sfârșit cât timp  $\{Golește Coadal2$  în  $Coadal\}$

cât timp  $Coadal2 \neq \emptyset$

$y \Leftarrow Coadal2$

$y \Rightarrow Coadal$

sfârșit cât timp

sfârșit dacă

*Return*

Numărul de operații 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 operații cerut egal cu  $2(|Coadal| - 1) + 1$  (primul ciclu) adunat cu  $2(|Coadal| - 1)$ , în total  $4|Coadal| - 3 = 4|Stiva| - 3$ .

Sugerăm cititorului să găsească și alte soluții pentru această problemă.

7. Algoritmii recursivi pentru parcurgeri sunt următorii:

(a) Parcurgerea în preordine se efectuează astfel: se prelucrează informația rădăcinii, apoi se prelucrează recursiv subarborele stâng, apoi subarborele drept.

*Preordine(Rad)*

dacă  $Rad \neq *$  atunci

scrie( *Info(Rad)* )  $\{$ Sau orice alt subalgoritm de prelucrare a informației din  $Rad\}$

*Preordine(LS(Rad))*

*Preordine(LD(Rad))*

sfârșit 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 informația din rădăcină.

Postordine(*Rad*)

dacă *Rad*  $\neq$  \* atunci

Postordine(*LS(Rad)*)

Postordine(*LD(Rad)*)

scrie( *Info(Rad)* ) {Sau orice alt subalgoritm de prelucrare a informației din *Rad*}

sfârșit dacă

Return

Apelul se face ce în cazul anterior.

(c) Parcurgerea recursivă în inordine se efectuează astfel: se parcurge recursiv subarborele stâng, apoi se prelucerează informația 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 prelucrare a informației din *Rad*}

Inordine(*LD(Rad)*)

sfârșit dacă

Return

Apelul se face ce în cazul anterior

### ***M1.U5.3. Test de autoevaluare a cunoștințelor***

1. Să se calculeze recursiv maximul unui șir.
2. Să se calculeze recursiv suma elementelor dintr-un șir.
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 înmulțiri.
5. Să se scrie un algoritm care calculează termenul *fibon*(*n*) al șirului Fibonacci în timp mai bun de  $\Theta(n)$  (a se vedea și problema 3).
6. Să se dea algoritmi recursivi pentru parcurgerea în preordine, postordine, inordine a unui arbore binar.

Problema 1. Defnim maximul unui șir  $x = (x(1), x(2), \dots, x(n))$  sub formă recursivă astfel:

$\text{maxim}(x(1 \dots n)) = \text{maxim}(x(1 \dots n)) = x(1)$ , dacă  $n=1$

$\text{maxim}(x(1 \dots n)) = \max\{x(n), \text{maxim}(x(1 \dots n-1))\}$ , dacă  $n > 1$

unde  $\max\{a, b\}$  este maximul dintre *a* și *b*. Algoritmul este dat mai jos:

Start MaximRec

*citește*( *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*

$MaximRec = \max\{x(n), Maxim(x, n - 1)\}$  *sfârșit dacă*

*Return*

Complexitatea este dată de ecuația  $T(n) = T(n-1) + c$  care are soluția

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

2. Problema 2. Definim suma elementelor unui șir  $x = (x(1), \dots, x(n))$  sub formă recursivă astfel:

$suma(x(1 \dots n)) = 0$ , dacă  $n=0$

$suma(x(1 \dots n)) = x(n) + suma(x(1 \dots n - 1))$ , dacă  $n > 0$

*Start SumaRec*

*citește* ( $n, (x(i), i = 1, n)$ )

$S = Suma(x, n)$

*scrie* ( $S$ )

*Stop*

unde subalgoritmul recursiv *Suma* este:

$Suma(x, n)$

*dacă*  $n = 0$  *atunci*

150

$Suma = 0$

*altfel*

$Suma = x(n) + Suma(x, n - 1)$

*sfârșit dacă*

*Return*

Complexitatea este dată de ecuația  $T(n) = T(n-1) + c$  care are soluția

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

3. Problema 3. Vom da o formulă recursivă de calcul a puterii naturale a unui număr.

$a^n = 1$ , dacă  $n = 0$

$a^n = a$ , dacă  $n = 1$

$a^n = (a^{n/2})^2$ , dacă  $n > 1$ ,  $n$  par

$a^n = a * (a^{n/2})^2$ , dacă  $n > 1$ ,  $n$  impar

Subalgoritmul recursiv este:

$Putere(a, n)$

*dacă*  $n = 0$  *atunci*

$Putere = 1$

*Return*

*sfârșit dacă*

*dacă*  $n = 1$  *atunci*

$Putere = a$

*Return*

*sfârșit dacă*

$temp = Putere(a, n/2)$

*dacă*  $n \bmod 2 = 0$  *atunci*  $\{n \text{ este par}\}$

$Putere = temp$

*altfel*

$Putere = a * temp$

*sfârșit dacă*



*Return*

Complexitatea este dată de ecuația  $T(n) = T(n/2) + c$ , de unde  $T(n) = \Theta(\log n)$

Problema 4. Pentru  $a \geq 1$ , conform algoritmului precedent, avem:

$$a^{15} = (a^{n/2})^{**}a = a^7 * a = (a^3)^2 * a = ((a)^2 * a)^2 * a$$

deci în total 4 înmulțiri.

Altfel, evident avem 6 înmulțiri.

Problema 5.

Se va aplica o metodă de tabelare a rezultatelor.

6. Problema 6. Algoritmii recursivi pentru parcurgeri sunt următorii:

(a) Parcurgerea în preordine se efectuează astfel: se prelucerează informația rădăcinii, apoi se prelucerează recursiv subarboarele stâng, apoi subarboarele drept.

*Preordine(Rad)*

*dacă Rad = \* atunci*

*scrie( Info(Rad) )* {Sau orice alt subalgoritm de prelucrarea a informației din Rad}

*Preordine(LS(Rad))*

*Preordine(LD(Rad))*

*sfârșit dacă*

*Return*

Apelul acestui subalgoritm se face cu *Preordine(Rad)*, unde *Rad* reprezintă rădăcina subarboarelui.

(b) Parcurgerea recursivă în postordine se efectuează astfel: se parcurge recursiv subarboarele stâng, apoi subarboarele drept, apoi se prelucerează informația din rădăcină.

*Postordine(Rad)*

*dacă Rad = \* atunci*

*Postordine(LS(Rad))*

*Postordine(LD(Rad))*

*scrie( Info(Rad) )* {Sau orice alt subalgoritm de prelucrarea a informației din Rad}

*sfârșit dacă*

*Return*

Apelul se face ce în cazul anterior.

(c) Parcurgerea recursivă în inordine se efectuează astfel: se parcurge recursiv subarboarele stâng, apoi se prelucerează informația din rădăcină, apoi se parcurge recursiv subarboarele drept.

*Inordine(Rad)*

*dacă Rad = \* atunci*

*Inordine(LS(Rad))*

*scrie( Info(Rad) )* {Sau orice alt subalgoritm de prelucrarea a informației din Rad}

*Inordine(LD(Rad))*

*sfârșit dacă*

*Return*

Apelul se face ce în cazul anterior.



### **M2.U1.3. Test de autoevaluare a cunoștințelor**

1. Scrieți un algoritm pentru parcurgerea tablei de șah cu un cal (calul sare pe diagonală unui dreptunghi cu laturile de un pătrat și două patrate). Fiecare pătrat al tablei trebui să fie vizitat exact o singură dată.
2. Având 4 culori și o hartă cu  $n$  țări (dată print-o matrice de adiacență:  
 $a_{ij} = 1$  dacă țara  $i$  este vecină cu țara  $j$ , 0 altfel), să se coloreze harta astfel ca două țări vecine să nu aibe aceeași culoare (două țări se consideră a fi vecine dacă au o frontieră comună).
3. O organizație are în componența sa  $n$  bărbați și  $m$  femei. Să se scrie un algoritm care listează toate modalitățile în care se poate alcătui o delegație care să conțină cel puțin  $k$  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  $b(i)$  bucăți? Să se dea toate soluțiile posibile.

### Răspunsuri:

**1.** Vom da două variante de rezolvare: iterativă și recursivă. Ambele variante vor folosi doi vectori de sărituri: *sarituraI* și *sarituraJ* cu câte 8 elemente, reprezentând cele 8 deplasări de la poziția curentă: de exemplu, dacă poziția curentă este pe linia  $i$  și coloana  $j$ , atunci prima săritură va duce calul la coordonatele  $(i+sarituraI(1), j+sarituraJ(1))$  (cu condiția 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 reține pentru fiecare celulă de coordonate  $(i, j)$  indicele săriturii care a determinat atingerea poziției respective (vom putea ști astfel de unde s-a sărit în poziția curentă). Soluția se dă sub forma unei matrice pătratice de ordinul  $n$  conținând în fiecare celulă de coordonate  $(i, j)$  indicele săriturii ce se face în continuare (cu excepția ultimei celule în care se sare).

*Start CalSah*

*citește*(  $n$  )

*sarituraI*(1) = +1, *sarituraJ*(1) = +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*(6) = +1

*sarituraI*(7) = +2, *sarituraJ*(7) = -1

*sarituraI*(8) = -2, *sarituraJ*(8) = -1

pentru  $i = 1, n$

pentru  $j = 1, n$

*deLa*( $i, j$ ) = 0

*tabla*( $i, j$ ) = 0

*sfârșit pentru*

*sfârșit pentru*

*nrSaritura* = 1

$i = j = 1$

cât timp  $nrSaritura > 0$

dacă  $nrSaritura = n2$  atunci

ScrieSolutie(tabla, n)

$indiceSarituraInapoi = deLa(i, j)$

$tabla(i, j) = 0$

$i = i - sarituraI(indiceSarituraInapoi)$

$j = j - sarituraJ(indiceSarituraInapoi)$

$nrSaritura = nrSaritura - 1$

sfârșit dacă

dacă  $tabla(i, j) < 8$  atunci

$tabla(i, j) = tabla(i, j) + 1$

$iInainte = i + sarituraI(tabla(i, j))$

$jInainte = j + sarituraJ(tabla(i, j))$

dacă PoateSari( $iInainte, jInainte, tabla, n$ )=adevarat atunci

$deLa(iInainte, jInainte) = tabla(i, j)$

$i = iInainte$

$j = jInainte$

$nrSaritura = nrSaritura + 1$

sfârșit dacă

altfel

dacă  $nrSaritura > 1$  atunci

$iInapoi = i - sarituraI(deLa(i, j))$

$jInapoi = j - sarituraJ(deLa(i, j))$

$tabla(i, j) = 0$

$i = iInapoi$

$j = jInapoi$

sfârșit dacă

$nrSaritura = nrSaritura - 1$

sfârșit dacă

sfârșit cât timp

Stop

ScrieSolutie(tabla, n)

pentru  $i = 1, n$

pentru  $j = 1, n$

scrie( tabla(i, j) )

sfârșit pentru

sfârșit pentru

Return

PoateSari(linie, coloana, tabla, n)

dacă  $linie > 0$  și  $linie < n + 1$  și  $coloana > 0$  și  $coloana < n + 1$  și  
 $tabla(linie, coloana) = 0$  atunci

rezultat = adevarat

altfel

rezultat = fals

sfârșit dacă

PoateSari = rezultat

Return

(b) Varianta recursivă

Cal( $i, j, n, nrSaritura, tabla$ )

dacă  $nrSaritura = n \cdot n$  atunci

ScrieSolutie( $tabla, n$ )

altfel

pentru  $indiceSaritura = 1, 8$

$iInainte = i + sarituraI[indiceSaritura]$

$jInainte = j + sarituraJ[indiceSaritura]$

dacă PoateSari( $iUrmator, jUrmator, tabla, n$ )=adev atunci

$tabla[iInainte, jInainte] = nrSaritura + 1$

Cal( $iInainte, jInainte, n, nrSaritura + 1, tabla$ )

$tabla[iInainte, jInainte] = 0$

sfârșit dacă

sfârșit pentru

sfârșit dacă

Return

Subalgoritmii ScrieSolutie și PoateSari sunt identici cu cei de la varianta iterativă.

Start CalSah

citește( $n$ )

$sarituraI(1) = +1, sarituraJ(2) = +2$

$sarituraI(2) = -1, sarituraJ(2) = +2$

$sarituraI(3) = +1, sarituraJ(3) = -2$

$sarituraI(4) = -1, sarituraJ(4) = -2$

$sarituraI(5) = +2, sarituraJ(5) = +1$

$sarituraI(6) = -2, sarituraJ(2) = +1$

$sarituraI(7) = +2, sarituraJ(2) = -1$

$sarituraI(1) = -2, sarituraJ(2) = -1$

pentru  $i = 1, n$

pentru  $j = 1, n$

$tabla(i, j) = 0$

sfârșit pentru

sfârșit pentru

$tabla(1, 1) = 1$

$Cal(1, 1, n, 1, tabla)$

Stop

2. S-a demonstrat cu ajutorul calculatorului că 4 culori sunt suficiente pentru a colora cu restricțiile date orice hartă. Vom da două variante, iterativă și recursivă. În ambele variante vectorul *culoare* de  $n$  elemente va conține culorile țărilor.

(a) Varianta iterativă.

*Start ColorareHarti*

citește(  $n$  )

citește(  $((a(i, j), j = 1, n), i = 1, n)$  )

pentru  $i = 1, n$

$culoare(i) = 0$

sfârșit pentru

$numarTara = 1$

cât timp  $numarTara > 0$

dacă  $numarTara = n + 1$  atunci

$scrie( culoare(i), i = 1, n )$

$numarTara = numarTara - 1$

sfârșit dacă

dacă  $culoare(numarTara) < 4$  atunci

$culoare(numarTara) = culoare(numarTara) + 1$

dacă  $BineColorat(a, numarTara, culoare) = adev$  atunci

$numarTara = numarTara + 1$

sfârșit dacă

altfel

$culoare(numarTara) = 0$

$numarTara = numarTara - 1$

sfârșit dacă

sfârșit cât timp

Stop

Subalgoritmul *BineColorat*(*vecin*, *numarTara*, *culoare*) returnează adevărat dacă culoarea țării curente (*numarTara*) satisface restricțiile cerute față de țările anterior colorate.

*BineColorat*( $a, numarTara, culoare$ )

*corect* = *adevarat*

$i = 1$

cât timp *corect* = *adevarat* și  $i < numarTara$

dacă  $a(i, numarTara) = 1$  și  $culoare(i) = culoare(numarTara)$  atunci

*corect* = *fals*

altfel

$i = i + 1$

sfârșit dacă

sfârșit 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

Cheama Coloreaza(*a*, *n*, *culoare*, *numarTara* + 1)

sfârșit dacă

sfârșit pentru

sfârșit dacă

Return

Subalgoritmul *BineColorat* este identic cu cel de la varianta iterativă.

Utilizarea subalgoritmului recursiv se face astfel:

*Start* ColorareHarti

citește( *n* )

citește( ((*vecin*(*i*, *j*), *j* = 1, *n*), *i* = 1, *n*) )

Cheama Coloreaza(*vecin*, *n*, *culoare*, 1)

Stop

**3.** Vom nota bărbații cu  $b_1, \dots, b_n$  iar femeile cu  $f_1, \dots, f_m$ . Vom genera delegațiile folosind doi vectori, unul pentru indicii de bărbați, celălalt pentru indicii de femei (vectorii *vb*, respectiv *vf*). Fiecare vector are o componență 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 delegații în care să nu existe nici un bărbat.

Varianta dată mai jos este recursivă. Pentru subalgoritmul *GenerareDelegatii*, parametrii *fc* și *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 toți indicii pentru femei}

dacă *bc* = *q* + 1 atunci {am generat toți indicii pentru bărbați}

scrie( (*vf*(*i*), *i* = 1, *p*) )

scrie( (*vb*(*i*), *i* = 1, *q*) )

altfel

pentru  $i = vb(bc - 1) + 1, n - q + bc$   
 $vb(bc) = i$   
Cheama GenerareDelegatii( $p, fc, m, q, bc + 1, n, vf, vb$ )  
sfârșit pentru  
sfârșit dacă

altfel

pentru  $i = vf(fc - 1) + 1, m - p + fc$   
 $vf(fc) = i$   
Cheama GenerareDelegatii( $p, fc + 1, m, q, bc, n, vf, vb$ )  
sfârșit pentru  
sfârșit dacă

Return  
Start Delegatii  
citește(  $m, n, k$  )  
 $vf(0) = vb(0) = 0$   
pentru  $p = k, m$   
pentru  $q = 0, n$   
Cheama GenerareDelegatii( $p, 1, m, q, 1, n, vf, vb$ )  
sfârșit pentru  
sfârșit pentru

Stop

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  $rest > 0$  și  $bc \leq n$  atunci  
pentru  $k = 0, b(bc)$   
 $solutie(bc) = k$   
Cheama Plata( $rest - k \cdot v(bc), solutie, bc + 1$ )  
sfârșit pentru  
sfârșit dacă

sfârșit dacă

Return  
Start PlataSuma  
citește(  $n, S$  )  
citește(  $b(i), i = 1, n$  )  
citește(  $v(i), i = 1, n$  )

Cheama Plata( $S, n, \text{solutie}, 1$ )

Stop

*Observație* Orice problemă cu datele ca mai sus poate fi transformată într-una în care vectorul  $b$  conține doar valoarea 1, prin crearea unui vector  $v\_$  care va conține valoarea  $v(i)$  de  $b(i)$  ori.



### ***M2.U2.3. Test de autoevaluare a cunoștințelor***

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

**Răspunsuri:**

**1. (a) Produs cartezian:**

CartezianRecursiv( $m, n, k, x$ )

dacă  $k = n + 1$  atunci

Cheama Afisare( $x, m$ ) {sau orice subalgoritm care prelucrează informația obținută}  
altfel

pentru  $p = 1, n(k)$

$x(k) = p$

Cheama CartezianRecursiv( $m, n, k + 1, x$ )

sfârșit pentru

sfârșit dacă

Return

Subalgoritmul Afisare este:

Afisare( $a, l$ )

pentru  $i = 1, l$

scrie(  $a(i)$  )

sfârșit pentru

Return

Programul principal este:

*Start* Cartezian

citește(  $m, (n(i), i = 1, m)$  )

Cheama CartezianRecrsiv( $m, n, 1, x$ )

Stop

Subalgoritmul recursiv generează soluțiile în ordine lexicografică.

**(b) Submulțimi:**

Vom folosi un subalgoritm recursiv cu forma: SubmultimiRecursiv( $n, k, cate, x$ ) unde  $n$  este dată de intrare reprezentând numărul elementelor mulțimii inițiale,  $k$  este indicele curent pentru care se încearcă generarea,  $cate$  reprezintă numărul de elemente ale submulțimii care se generează ( $1 \leq k \leq cate$ , pentru  $cate > 0$ ), iar  $x$  este vectorul soluție. Vom considera că  $x$  are un element de indice 0 care va avea permanent valoarea 0.



SumbultimiRecurziv( $n, k, cate, x$ )

dacă  $k > cate$  atunci

Cheama ScrieSolutie( $x, cate$ ) {sau orice subalgoritm care prelucrează informația dată}

altfel

pentru  $p = x(k - 1), n - cate + k$

$x(k) = p$

Cheama SubmultimiRecurziv( $n, k + 1, cate, x$ )

sfârșit pentru

sfârșit dacă

Return

Subalgoritmul *ScrieSolutie* este o variație a celui anterior, care în plus ia în considerare cazul în care s-a generat mulțimea vidă:

ScrieSolutie( $a, l$ )

dacă  $l = 0$  atunci

scrie( “mulțimea vidă” )

altfel

pentru  $i = 1, l$

scrie(  $a(i)$  )

sfârșit pentru

sfârșit dacă

Return

Subalgoritmul principal este:

Start Submultimi

citește(  $n$  )

$x(0) = 0$

pentru  $cate = 0, n$

Cheama SubmultimiRecurziv( $n, 1, cate, x$ )

sfârșit pentru

Stop

Subalgoritmul recursiv generează submulțimile în ordine lexicografică.

### (c) Combinări:

Subalgoritmul este același cu cel de la generarea recursivă a submulțimilor, cu aceeași semnificație a parametrilor de apel. Programul principal este:

Start Combinari

citește(  $n, m$  )

$x(0) = 0$

Cheama CombinariRecurziv( $n, 1, m, x$ )

Stop

Subalgoritmul recursiv generează combinațiile în ordine lexicografică.

**(d) Permutări:**

PermutariRecurziv( $k, x, n$ )

dacă  $k = 0$  atunci

Cheama ScribeSolutie( $v, n$ )

altfel

pentru  $i = 1, k$

$v(k) \leftrightarrow v(i)$

Cheama PermutariRecurziv( $k - 1, x, n$ )

$v(k) \leftrightarrow v(i)$

sfârșit pentru

sfârșit dacă

Return

Subalgoritmul de *ScribeSolutie* este cel de la produs cartezian.

Programul principal ar fi:

*Start* Permutari

citește( $n$ )

pentru  $i = 1, n$

$v(i) = i$

sfârșit pentru

Cheama PermutariRecurziv( $n, x, n$ )

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



**M2.U3.3. Test de autoevaluare a cunoștințelor**

1. Bazându-vă pe observația că mutarea din mijloc se știe, scrieți un algoritm iterativ pentru problema turnurilor din Hanoi, punând mereu mijloacele intervalelor obținute prin înjumătățire într-un șir cu numărul cunoscut de mutări.
2. Având la intrare un șir ordonat crescător, să se scrie un algoritm de tip “Divide et Impera” pentru găsirea poziției unui element de valoare dată  $x$ , dacă el există. Se va folosi metoda înjumătățirii intervalului dat de numărul de elemente al șirului. Care este complexitatea căutării?
3. Rezolvați aceeași problemă prin căutare secvențială (testarea pe rând a elementelor). Care este în acest caz complexitatea?
4. Demonstrați că timpul de execuție 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, partiționarea se face cu un singur element într-una din partiții. Rescrieți algoritmul de partiționare, luând ca element de separare nu primul element, ci valoarea din mijloc între primul element, ultimul și cel aflat pe poziția din mijloc.

### Răspunsuri:

1. Vom proceda în felul următor: dacă trebuie să umplem un vector al cărui capăt stâng este  $st$  și 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 recursivității. Soluțiile vor fi memorate într-o matrice cu  $2n - 1$  linii și 2 coloane.

Start Hanoi

citește(  $n$  )

$i = 1$

$j = 2$

$st = 1$

$dr = 2n - 1$

$S = \emptyset$  {S este stivă}

$(i, j, st, dr) \Rightarrow S$

cât timp  $S \neq \emptyset$

$(i, j, st, dr) \Leftarrow S$

$sol(st+dr/2, 1) = i$

$sol(st+dr/2, 2) = j$

dacă  $st \leq st+dr/2 - 1$  atunci

$(i, 6 - i - j, st, st+dr/2 - 1) \Rightarrow S$

sfârșit dacă

dacă  $st+dr/2 + 1, dr$  atunci

$(6 - i - j, j, st+dr/2 + 1, dr) \Rightarrow S$

sfârșit dacă

sfârșit cât timp

scrie(  $sol(i, 1), sol(1, 2), i = 1, 2n - 1$  )

Stop

Adâncimea stivei  $S$  este  $\Theta(\log_2(2n-1)) = \Theta(n)$ , iar complexitatea este evident  $\Theta(2^n)$ .

2. Complexitatea obținută este  $T(n) = \Theta(\log n)$ .

4. La fiecare pas din ciclul *repetă* al algoritmului de partiționare  $i$  crește cu o unitate, iar  $j$  scade cu o unitate. Partiționarea se oprește atunci când  $j$  este la jumătatea șirului, deci împărțirea este echilibrată. Ca atare, complexitatea este dată de  $T(n) = T(n/2) + c \cdot n$ , de unde  $T(n) = \Theta(n \cdot \log n)$ .

5. Vom crea o funcție care determină dacă un număr este sau nu situat între alte două numere:  
 $\text{Intre}(x, y, z)$  {returnează adevărat dacă  $x$  este între  $y$  și  $z$ , fals altfel}

dacă ( $y \leq x$  și  $x \leq z$ ) sau ( $z \leq x$  și  $x \leq y$ ) atunci  
     $Intre = adevarat$

altfel

Return fals

sfârșit dacă

Algoritmul de partiționare modificat este:

Partitie( $a, p, q$ )

$r = p + q$

dacă Intre( $a(p), a(q), a(r)$ )) atunci

$indiceMediana = p$

sfârșit dacă

dacă Intre( $a(q), a(p), a(r)$ )) atunci

$indiceMediana = q$

sfârșit dacă

dacă Intre( $a(r), a(p), a(q)$ ) atunci

$indiceMediana = r$

sfârșit dacă

dacă  $p \neq indiceMediana$  atunci

$a(p) \leftrightarrow a(indiceMediana)$

sfârșit 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

$k = j$

$terminat = adevarat$

sfârșit dacă

până când  $terminat = adevarat$

Return



### ***M2.U4.3. Test de autoevaluare a cunoștințelor***

1. Descrieți un algoritm care, fiind dată o mulțime  $S$  de  $n$  numere întregi și distincte și un întreg pozitiv  $k \leq n$ , determină cele  $k$  numere care sunt cele mai apropiate de mediana lui  $S$ .
2. Fie  $a[1 \dots n]$ ,  $b[1 \dots n]$  două șiruri sortate. Scrieți un algoritm performant care găsește mediana celor  $2n$  numere.
3. Fiind dată o mulțime de  $n$  numere, dorim să găsim cele mai mari  $i$  numere în ordinea sortată, folosind un algoritm bazat pe comparații. Să se scrie mai multe variante și să se compare din punct de vedere al complexității.
4. Să se modifice algoritmul CountSort astfel încât să sorteze și șiruri conținând numere întregi negative.

### **Răspunsuri**

1. Rezolvarea problemei se face în doi pași:

- (a) Se determină mediana șirului inițial  $x = (x(1), \dots, x(n))$ ; fie ea  $medx$
- (b) Se calculează șirul auxiliar  $y = (y(i), i = 1, n)$  unde  $y(i) = |x(i) - medx|$
- (c) Se determină a  $k$ -a statistică de ordine a șirului  $y$ ; fie ea  $medy$
- (d) Se determină  $k$  elemente mai mici sau egale cu  $medy$ ; indicii acestor elemente sunt indicii elementelor căutate din șirul  $x$ .

Complexitatea fiecărui pas de mai sus este  $\Theta(n)$ , deci complexitatea algoritmului schițat este  $\Theta(n)$ . Subalgoritmul în pseudocod este:

*Start* Apropiate

*citește*(  $n, k, (x(i), i = 1, n)$  )

Cheama Selectie( $x, 1, n, k, medx$ )

Cheama ObtineY( $x, n, y, medx$ )

Cheama Selectie( $x, 1, n, k, marginey$ )

Cheama Vecini( $x, y, n, marginey, z$ )

*Stop*

Subalgoritmul Selectie este cel care determină a  $k$ -a statistică de ordine a unui șir (cu complexitate liniară pentru cazul cel mai defavorabil); subalgoritmul ObtineY este:

ObtineY( $x, n, y, medx$ )

pentru  $i = 1, n$

$y(i) = |x(i) - medx|$

sfârșit pentru

Return

Subalgoritmul Vecini va determina cei mai apropiați  $k$  vecini ai șirului  $x$  față de mediana sa:

Vecini( $x, y, n, marginey, z$ )

$lungZ = 0$

pentru  $i = 1, n$

dacă  $y(i) \leq \text{marginay atunci}$

$\text{lungZ} = \text{lungZ} + 1$

$z(\text{lungZ}) = x(i)$

sfârșit dacă

sfârșit pentru

Return

2. Vom folosi faptul că cele două șiruri sunt sortate și au aceeași lungime. De asemenea ne folosim de observația evidentă că mediana unui șir sortat se determină în timp constant: dacă  $i$  este indicele primului element din șir, iar  $j$  este indicele ultimului element al șirului, atunci mediana este pe poziția  $k = (i+j)/2$

Pentru două șiruri  $A(i_A \dots j_A)$  și  $B(i_B \dots j_B)$  cu același 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 și obținem mediana lor (timp  $\Theta(1)$ ). Dacă numărul de elemente este mai mare decât 2, atunci fie  $k_A$  respectiv  $k_B$  indicii mijloacelor vectorilor  $A(i_A \dots j_A)$  și  $B(i_B \dots j_B)$ . Dacă  $A(k_A) < B(k_B)$  atunci vom ignora elementele din  $A(i_A \dots j_A)$  aflate la stânga indicelui  $k_A$  și elementele din  $B(i_B \dots j_B)$  aflate la dreapta indicelui  $k_B$  și vom considera doar vectorii  $A(k_A \dots j_A)$  și  $B(k_B \dots j_B)$ . Analog se procedează dacă  $A(k_A) \geq B(k_B)$ . Procesul se reia pentru vectorii înjumătățiți.

Justificăm faptul că prin eliminarea unei jumătăți 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  $A(i) \leq A(k_A) \leq A(p) \quad \forall p = k_A, \dots, j_A, A(i) \leq A(k_A) < B(k_B) \leq B(q) \quad \forall q = k_B, \dots, j_B$ .

Obținem că  $A(i)$  este mai mic sau egal decât  $2(n - \left\lfloor \frac{n-1}{2} \right\rfloor) + 1$  Dar  $2(n - \left\lfloor \frac{n-1}{2} \right\rfloor) + 1 \geq n+1$ , deci

$A(i)$  este mai mic sau egal decât cel puțin  $n + 1$  elemente. Ca atare, ignorându-le, nu pierdem mediana. Analog se demonstrează că eliminând jumătatea dreaptă a lui  $B$  nu se pierde mediana. Demonstrația este asemănătoare pentru cazul în care  $A(k_A) \geq B(k_B)$ .

Algoritmul este dat mai jos:

Start MedianaSiruri

citește  $(n, (A(i), B(i), i = 1, n))$

$iA = iB = 1$

$jA = jB = n$

cât timp  $iA < iB$

$kA = (iA + jA)/2$

$kB = (iB + jB)/2$

$n = n/2$

dacă  $A(kA) < B(kB)$  atunci

$iA = kA$

$jB = kB$

altfel

$$jA = kA$$

$$iB = kB$$

sfârșit dacă

sfârșit cât timp

scrie(  $\max(A(iA), B(iB))$  )

Stop

unde max returnează maximumul a două numere. Complexitatea este:

$$T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$$

3. Se poate proceda în mai multe moduri:

(a) Sortăm numerele și alegem cele mai mari  $i$  numere

(b) Construim un (max)heap și extragem din el de  $i$  ori maximumul

(c) Prin utilizarea unui algoritm de statistică de ordine

Lăsăm în seama cititorului scrierea algoritmilor pentru cazurile 3a și 3b. Pentru situația 3c determinăm a  $n - i + 1$ -a statistică de ordine (fie ea  $x$ ) și apoi printr-o parcurgere a șirului numerele mai mari sau egale cu  $x$ . Complexitatea este de  $\Theta(n + i \cdot \log i)$ . Sugerăm cititorului compararea acestei complexități cu cea obținută la punctele 3a și 3b.

4. Ideea este de a “deplasa” elementele de-a lungul axei astfel încât să devină toate pozitive (caz pentru care algoritmul Countsort funcționează). Această deplasare se obține 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 cantității adunate anterior.

*Start CountSortModificat*

citește(  $n, (x(i), i = 1, n)$  )

$min = \text{MinimSir}(x, n)$  {Funcție care determina minimumul unui sir}

pentru  $i = 1, n$

$$x(i) = x(i) + (-min)$$

sfârșit pentru

Cheama CountSort

pentru  $i = 1, n$

$$y(i) = x(i) - (-min)$$

sfârșit pentru

Stop



### ***M2.U5.3. Test de autoevaluare a cunoștințelor***

1. Dacă  $x = (x(i), i = 1, m)$ ,  $y = (y(i), i = 1, n)$  reprezintă două mulțimi de elemente, atunci să se determine mulțimea intersecție  $z = (z(i), i = 1, k)$ .

2. Dându-se  $n$  obiecte de cost  $c = (c(i), i = 1, n)$  cu greutatea  $g = (g(i), i = 1, n)$  și 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 cunoaște intervalul în care se desfășoară:  $[st, sf)$ . Se cere să se planifice un număr maxim de spectacole astfel încât să nu se suprapună..

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

$$E = a_1 * x_1 + a_2 * x_2 + \dots + a_m * x_m \text{ unde } x_i \in B..$$

5. O stație de servire trebuie să satisfacă cererile a  $n$  clienți. Timpul de servire pentru fiecare client este cunoscut: pentru clientul  $i$  timpul este  $t_i$ . Să se minimizeze timpul total de așteptare

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

### Răspunsuri.

1. Pentru fiecare element  $x(i)$  se determină dacă apare și în șirul  $y$ . în caz afirmativ,  $x(i)$  va face parte și din mulțimea intersecție. Faptul că în acest fel se obține mulțimea intersecție este (sperăm) evident. Complexitatea algoritmului este dependentă de modul în care se face căutarea lui  $x(i)$  în  $y$ .

- Dacă se face căutare liniară atunci complexitatea este  $\Theta(mn)$

- Dacă se caută binar, 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 obține o complexitate de  $\Theta((m + n) \log m)$ . Este mai avantajos ca să se facă sortarea și căutarea în șirul cel mai scurt. Lăsăm cititorul să conceapă algoritmul pentru rezolvarea acestei probleme. În ceea ce privește funcția care se optimizează, este vorba de:  $\max f(x, y) = |\{a : a \text{ în } x \text{ și } a \text{ în } y\}|$  ( $|A|$  reprezintă numărul de elemente ale mulțimii  $A$ ), care se regăsește și în algoritmul prezentat (cu toate că nu este evident).

2. Ideea algoritmului este de a calcula pentru fiecare obiect

valoarea lui unitară, *i.e.* cât se câștigă din transportarea unei unități din obiectul respectiv. Suntem mai interesați în a considera obiectele profitabile (preț mare, greutate mică) înaintea celor pentru care raportul preț/greutate este mic. Algoritmul dat mai jos sintetizează această strategie.

*Start RucsacContinuu*

citește(  $n, G, (c(i), g(i), i = 1, n)$  )

pentru  $i = 1, n$

$$v(i) = c(i)/g(i)$$

sfârșit pentru

Cheama 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 pux în rucsac}

$S = 0$  { $S$  este suma realizată prin luarea obiectelor}

cât timp  $k \geq n$  și  $gc < G$

dacă  $gc + g(k) \geq G$  atunci



$f(k) = 1$   $\{f(k)$  reprezintă fracțiunea care se ia din obiectul  $k\}$

$$gc = gc + g(k)$$

$$S = S + c(k)$$

$$k = k + 1$$

altfel

$$f(k) = (G - gc) / g(k)$$

$$gc = G$$

$$S = S + f(k)g(k)$$

sfârșit dacă

sfârșit cât timp

dacă  $k > n$  atunci

$k = n$  {în cazul în care  $G$  este mai mare sau egal cu greutatea tuturor obiectelor}

sfârșit dacă

pentru  $i = 1, k$

scrie(  $f(i)$  )

sfârșit pentru

Stop

Subalgoritmul *Sortare*( $c, g, v, n$ ) rearanjează elementele vectorilor  $c, g, v$

astfel încât  $v(1) \geq v(2) \geq \dots \geq v(n)$ . Corectitudinea algoritmului de

mai sus este dată de propoziția de mai jos:

**Propoziția 1.** Algoritmul precedent determină câștigul maxim în restricțiile impuse problemei.

*Demonstrație.* Dacă  $G > \sum_{i=1}^n g(i)$ , atunci în mod evident soluția dată de vectorul  $f$  va avea forma

$f = (1, 1, \dots, 1)$  (se iau toate obiectele), ceea ce evident înseamnă optim. Vom considera deci cazul în care

$G < \sum_{i=1}^n g(i)$ . Este ușor de văzut că soluția dată de vectorul  $f$  are forma:

$$f = (1, 1, \dots, 1, \varepsilon, 0, 0, \dots, 0) \quad (\text{B.1})$$

unde  $0 < \varepsilon \leq 1$  și  $\varepsilon$  apare pe poziția  $k$  ( $k$  fiind variabila din algoritm).

$$\text{Avem că: } S = \sum_{i=1}^n f(i)c(i) \quad (\text{B.2})$$

și

$$G = \sum_{i=1}^k f(i)c(i) \quad (\text{B.3})$$

Fie  $S_{\max}$  câștigul maxim care se poate realiza în condițiile problemei și fie

$$F' = (f'(1), \dots, f'(n)) \quad (\text{B.4})$$

fracțiunile din obiecte care se iau, pentru a realiza  $S'$ :

$$S' = \sum_{i=1}^n f'(i)c(i) \quad (\text{B.5})$$

Avem:

$$G' = \sum_{i=1}^k f'(i)c(i) \quad (\text{B.6})$$

unde  $G'$  este greutatea totală care se transportă pentru câștigul  $S'$ .

Putem avea următoarele situații:

- (a) Dacă (prin absurd)  $S' < S$ , atunci se contrazice optimalitatea lui  $S'$ ; deci această situație nu poate să apară.
- (b) Dacă  $S' = S$ , atunci  $S$  este optimal și 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  $f''(p) = f'(p) + \alpha = 1$ , obținând un surplus de greutate de  $\alpha g(p)$  (depășim 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

$$\beta g(t) = \alpha g(p) \quad (\text{B.7})$$

Pentru  $i \neq p, t$  vom avea  $f''(i) = f'(i)$ . Atunci:

$$S'' = \sum_i f''(i)c(i) = \sum_{i \neq p, t} f'(i)c(i) + (f'(p) + \alpha)c(p) + (f'(t) - \beta)c(t) = S' + \alpha c(p) - \beta c(t) \quad (\text{B.8})$$

Dar:  $\alpha c(p) - \beta c(t) = \alpha v(p)g(p) - \beta v(t)g(t)$  și conform relației (B.7) obținem că:

$$S'' = S' + \alpha g(p)(v(p) - v(t)) \quad (\text{B.9})$$

Dacă  $v(p) - v(t) = 0$ , atunci  $S' = S''$  și 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 relația dintre  $f(k) = \varepsilon$  și  $f'(k)$ . Dacă  $f'(k) < \varepsilon$  atunci printr-un procedeu analog de mărire a lui  $f'(k)$  la  $\varepsilon$  (și scăderea unui  $f'(t)$ ,  $t > k$ ) vom obține fie o contradicție,

fie  $S'' = S'$  (ca mai sus). Dacă  $f'(k) > \varepsilon$ , atunci se contrazice modul de alegere al lui  $f'(k) = \varepsilon$  din algoritm. În sfârșit, dacă  $f'(k) = \varepsilon$  atunci avem că  $f'$  poate fi adus la forma  $f$  fără a pierde din câștig. Deci am demonstrat că  $S$  dat de  $f$  (conform algoritmului) nu poate fi mai mic decât  $S'$ , de unde rezultă propoziția.

**3. Algoritmul 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 șir ultimului spectacol programat care îndeplinește condiția că începe după ce s-a terminat ultimul spectacol programat.
- (d) dacă tentativa de mai sus a eșuat (nu am găsit un astfel de spectacol), algoritmul se termină, altfel se programează spectacolul găsit și se reia pasul 3c.

Algoritmul este:

*Start* Spectacole

citește(  $n$ , ( $st(i)$ ,  $sf(i)$ ,  $i = 1, n$ ) )

Cheama Sortare( $st$ ,  $sf$ ,  $n$ )

scrie( “Spectacol care începe la ”,  $st(1)$ , “și se sfârșește la”,  $sf(1)$  )

$k = 1$

pentru  $i = 2, n$

dacă  $st(i) \geq sf(k)$  atunci

scrie( “Spectacol care începe la ”,  $st(i)$ , “și se sfârșește la”,  $sf(i)$  )

$k = i$

sfârșit dacă

sfârșit pentru

Stop

Algoritmul  $Sortare(st, sf, n)$  interschimbă elementele din  $st$ , respectiv  $sf$  astfel încât  $sf(1) \leq sf(2) \leq \dots \leq sf(n)$ .

Complexitate: avem o sortare care se poate face în timp  $\Theta(n \log n)$  și o parcurgere a șirului  $st$ , deci în total  $\Theta(n \log n)$ .

4. Deși problema are un caracter teoretic pronunțat, rezultatul obținut poate fi folosit în numeroase cazuri. Strategia pentru această problemă este:

(a) Se sortează elementele celor două mulțimi

(b) Dacă  $A = \{a_1, \dots, a_m\}$  are  $k$  elemente negative și  $p$  elemente pozitive, se aleg primele  $k$  elemente ale lui  $B$  și ultimele  $p$  elemente ale lui  $B$ .

Algoritmul este:

Start Maxim

citește(  $m, n, (a(i), i = 1, m), (b(i), i = 1, n)$  )

Cheama  $Sortare(a, m)$

Cheama  $Sortare(b, n)$

$k = 1$

cât timp  $k \leq m$  și  $a(k) < 0$

$k = k + 1$

sfârșit cât timp

$k = k - 1$

$suma = 0$

pentru  $i = 1, k$

$suma = suma + a(i) * b(i)$

sfârșit pentru

pentru  $i = k + 1, m$

$suma = suma + a(i) * b(n - m + i)$

sfârșit pentru

Stop

5. Timpul de așteptare pentru un client este compus din timpii de servire ai clienților care sunt procesați înaintea lui adunat cu timpul de servire pentru el însuși. De aceea timpul total de așteptare nu este indiferent față de ordinea de servire. De exemplu, dacă avem doi clienți, cu  $t_1 = 1$  și  $t_2 = 2$ , atunci servirea în ordinea 1, 2 duce la un timp de așteptare de  $1 + (1 + 2) = 4$ , pe

cîdacă ar fi procesați invers, s-ar ajunge la timpul  $2 + (2 + 1) = 5$ . Intuitiv, este mai bine dacă clienții sunt procesați 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 clienților (corespunzând unei ordini de intrare); clienții sunt preluați în ordinea  $p_1, p_2, \dots, p_n$ . Pentru aceasă ordine, timpul total de servire este:

$$T(p) = t_{p1} + (t_{p1} + t_{p2}) + \dots + (t_{p1} + t_{p2} + \dots + t_{pn}) = \sum_{i=1}^n (n - i + 1) t_{p_i} \quad (\text{B.10})$$

Fie  $p$  permutarea care determină  $T$  din (B.10) minim. Fie  $p'$  permutarea care se obține 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

citește(  $n$ , ( $t(i)$ ,  $i = 1, n$ ) )

pentru  $i = 1, n$

$p(i) = i$

sfârșit pentru

Cheama Sortare( $t, p, n$ )

pentru  $i = 1, n$

scrie(  $p(i)$  )

sfârșit pentru

Stop

Algoritmul *Sortare*( $t, p, n$ ) sortează elementele lui  $t$  făcând, în paralel, aceleași interschimbări în vectorul de permutare  $p$ .



### ***M2.U6.3. Test de autoevaluare a cunoștințelor***

1. Găsiți o parantezare optimă a produsului unui șir de matrice al cărui șir de dimensiuni este (5, 10, 3, 12, 5, 50, 6) (adică  $5 \times 10$ ,  $10 \times 3$ , etc).
2. Arătați că o parantezare completă a unei expresii cu  $n$  elemente are exact  $n - 1$  perechi de paranteze.
3. 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-și-la-dreapta*. Se cere un algoritm care găsește suma maximă.

Exemplu:

5

2 3

8 1 2

5 9 3 4

Suma maximă este  $S = 5+2+8+9$ .

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

### Răspunsuri

1. Tablourile  $m$  și  $s$ , analoage celor din unitatea de învățare 2.6

1	2	3	4	5	6	1	2	3	4	5	6
0	150	330	405	165 5	201 0	0	1	2	2	4	2
0	0	360	330	243 0	195 0	0	0	2	2	2	2
0	0	0	180	930	177 0	0	0	0	3	4	4
0	0	0	0	300 0	186 0	0	0	0	0	4	4
0	0	0	0	0	150 0	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 înmulțiri scalare este  $m(1, 6) = 2010$ . Pentru determinarea modului în care se face înmulțirea, folosim informația din matricea  $s$ :  $s(1, 6) = 2$ , deci vom avea parantezarea  $(A1 \cdot A2) \cdot (A3 \cdot \dots \cdot A6)$ . Pentru a determina cum se face parantezarea pentru produsul  $A_{3..6}$ , folosim  $s(3, 6) = 4$ , de unde deducem că grupăm  $A_{3..6} = (A3 \cdot A4) \cdot (A5 \cdot A6)$ . în final, obținem parantezarea completă:  $((A1 \cdot A2) \cdot ((A3 \cdot A4) \cdot (A5 \cdot A6)))$

2. Demonstrația se face prin inducție matematică după numărul  $n$  de matrici. Fie  $P(n)$  propoziția “parantezarea completă a unui șir cu  $n$  matrici are exact  $n - 1$  paranteze”.

Verificarea:

(a)  $P(1)$ : o matrice reprezintă o expresie complet parantezată

(b)  $P(2)$ :  $(A1 \cdot A2)$  este singura expresie complet parantezată asociată produsului a două matrici, deci avem o pereche de paranteze.

Demonstrație: presupunem propoziția adevărată până la  $n - 1$  și demonstrăm că și  $P(n)$  este adevărată. Fie  $A_{1..n} = (A_{1..p} \cdot A_{p+1..n})$ ,  $1 \leq p < n$ . Atunci: conform propoziției  $P(p)$  avem că numărul de perechi de paranteze pentru  $A_{1..p}$  este  $p - 1$ , iar pentru  $A_{p+1..n}$  este  $n - (p + 1)$ . Pentru  $A_{1..n}$  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 enunțul problemei.

3. Relația după care se calculează optimul este:

$$m(i, j) = \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j\} \quad (\text{B.11})$$

Fie  $\alpha \leq \beta$ .  $m(\alpha, \beta)$  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 și  $T(\alpha, \beta)$  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  $U(\alpha, \beta) = n - \beta$ .

$$\text{Obținem: } R(\alpha, \beta) = T(\alpha, \beta) = \sum_{\alpha=1}^n \sum_{\beta=\alpha}^n (n - \beta + \alpha - 1) = \dots = \frac{n^3 - n}{3}$$

4. Fie  $a = (a(i, j))$  matricea triunghiulară dată la intrare ( $1 \leq i \leq n$ ,  $1 \leq j \leq i$ ). Observăm că avem  $2n-1$  posibilități de a forma drumuri de la  $a(1, 1)$  la ultima linie (demonstrația se face prin inducție matematică după  $n$ ). Ca atare, o metodă de abordare de tip *brute force* este de la început sortită eșecului. De asemenea se observă că o strategie de tipul “coboară la numărul cel mai mare aflat pe linia imediat următoare” nu duce la soluție optimă. Vom da un algoritm bazat pe programare dinamică. Fie  $D$  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  $i$  și coloana  $j$ . Atunci  $S(i, j)$  este valoarea maximă care se poate obține plecând din  $(i, j)$  până la ultima linie. Dacă nu ar fi așa, atunci înseamnă că ar exista un alt drum de la  $(i, j)$  la ultima linie care dă valoare mai mare, iar prin modificarea lui  $D$  astfel încât să includă acest ultim drum am obține o valoare mai mare, contradicție cu alegerea lui  $D$ . Ca atare, putem deduce că:

$$S(i, j) = \begin{cases} S(n, j) & \text{daca } i = n \\ a(i, j) + \max\{S(i+1, j), S(i+1, j+1)\} & \text{daca } i < n \end{cases} \quad (\text{B.12})$$

pentru  $1 \leq i \leq n$ ,  $1 \leq j \leq i$ ; ceea ce ne duce la algoritm. Calculul lui  $S(i, j)$  nu se face recursiv, ci folosind o matrice triunghiulară. Pentru reconstituirea soluției se folosește o matrice triunghilară *sol*.

*Start Triunghi*

citește(  $n$ ,  $(a(i, j), i = 1, n, j = 1, i)$  )

pentru  $j = 1, n$

$s(n, j) = a(n, j)$

sfârșit 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

$s(i, j) = a(i, j) + s(i+1, j+1)$

$sol(i, j) = j + 1$

sfârșit dacă

sfârșit pentru

sfârșit pentru

scrie( “Suma maximă este”,  $s(1, 1)$  )

scrie( “Ea se obține astfel:” )

$i = j = 1$

scrie(  $a(1, 1)$  )

pentru  $k = 1, n - 1$

$j = sol(i, j)$

$i = i + 1$

scrie(  $a(i, j)$  )

sfârșit pentru

Stop

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  $Ai...j$  vom determina indicele  $k$  pentru care facem descompunerea  $Ai...j = (Ai...k) \cdot (Ak+1...j)$ , 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

sfârșit dacă

dacă  $j - i = 1$  atunci

scrie( “(”,  $i$ , “\*”,  $j$ , “)” )

Return

sfârșit dacă

$k = s(i, j)$

scrie( “(” )

Cheama Inmultiri( $s, i, k$ )

scrie( “)”\*(” )

Cheama Inmultiri( $s, k + 1, j$ )

scrie( “)” )

Return

Propunem cititorului interesat crearea unui algoritm care să determine *toate* modalitățile de înmulțire a matricelor astfel încât să se ajungă la un număr de înmulțiri scalare minim. (Indicație: nu se mai folosește informația 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 și analiza algoritmilor*. Editura Științifică și Enciclopedică, 1986.
- [4] Tudor, Sorin. *Tehnici de programare (Manual pentru clasa a X-a)*. L&S Infomat, 1996.