# ALGORITMI ŞI STRUCTURI DE DATE

Note de curs

(draft v1.1)

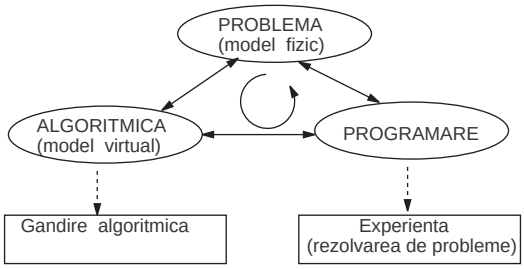
# Prefaţă

Când dorim să reprezentăm obiectele din lumea reală într-un program pe calculator, trebuie să avem în vedere:

- *modelarea* obiectelor din lumea reală sub forma unor entităţi matematice abstracte şi tipuri de date,
- *operaţiile* pentru înregistrarea, accesul şi utilizarea acestor entităţi,
- *reprezentarea* acestor entităţi în memoria calculatorului, şi
- *algoritmii* pentru efectuarea acestor operaţii.

Primele două elemente sunt în esenţă de natură matematică şi se referă la "ce" structuri de date şi operaţii trebuie să folosim, iar ultimile două elemente implică faza de implementare şi se referă la "cum" să realizăm structurile de date şi operaţiile. *Algoritmica* şi *structurile de date* nu pot fi separate. Deşi *algoritmica* şi *programarea* pot fi separate, noi nu vom face acest lucru, ci vom implementa algoritmii într-un limbaj de programare (Pascal, C/C++, Java). Din această cauză acest curs este şi o iniţiere în *algoritmică şi programare*.

Scopul cursului este subordonat scopului specializării (informatică, în cazul nostru) care este să pregătească specialişti **competenţi**, cu înaltă calificare în domeniul informaticii, cadre didactice **competente** în acest domeniu (profesor de informatică în gimnaziu şi liceu), informaticieni în diverse domenii cu profil tehnic, economic, etc. ce pot începe lucrul imediat după absolvirea facultăţii.Dezideratul final este deci **competenţa**. Competenţa într-un domeniu de activitate implică **experienţă** în **rezolvarea problemelor** din acel domeniu de activitate. Atât competenţa cât şi experienţa în rezolvarea problemelor se pot obţine numai dacă permanent se întreprind eforturi pentru *însuşirea de noi cunoştinţe*. De exemplu, orice informatician (programator sau profesor) care elaborează programe pentru rezolvarea unor probleme diverse, trebuie să aibă competenţe conform schemei[1]:




Cursul de *Algoritmi şi structuri de date* este util (şi chiar necesar) pentru **formarea competenţelor şi abilităţilor** unui bun programator sau profesor de informatică. Pentru a vedea care sunt aceste competenţe şi abilităţi putem, de

[1] M. Vlada; E-Learning şi Software educaţional; Conferinţa Naţională de Învăţământ Virtual, Bucureşti, 2003

exemplu, să citim *Programa pentru informatică - Concursul naţional unic pentru ocuparea posturilor didactice declarate vacante în învăţământul preuniversitar*.[2]

Într-un fel, primul semestru al cursului *Algoritmi şi structuri de date* este echivalent cu ceea ce se predă la informatică în clasa a IX-a iar al doilea semestru cu clasa a X-a (specializarea: matematică-informatică, intensiv informatică). Diferenţa este dată în primul rând de dificultatea problemelor abordate de către noi în cadrul acestui curs. Din această cauză vom avea în vedere şi ce prevede *Pograma şolară pentru clasa a IX-a, Profil real, Specializarea: Matematică-informatică, intensiv informatică*. De asemenea, merită să vedem ce păreri au cei care au terminat de curând o facultate cu un profil de informatică şi care au un început de carieră reuşit. Vom înţelege de ce acest curs este orientat pe **rezolvarea de probleme**.

Alegerea limbajului Java pentru prezentarea implementărilor algoritmilor a fost făcută din câteva considerente. Java verifică validitatea indicilor tablourilor (programele nu se pot termina printr-o violare de memorie sau eroare de sistem). Java realizează gestiunea automată a memoriei (recuperează automat memoria care nu mai este necesară programului) ceea ce simplifică scrierea programelor şi permite programatorului să se concentreze asupra esenţei algoritmului. Există documentaţie pe internet. Compilatorul de Java este gratuit. Un program scris în Java poate fi executat pe orice calculator (indiferent de arhitectură sau sistem de operare).

Studenţii **nu sunt obligaţi** să realizeze implementările algoritmilor în Java; ei pot folosi Pascal sau C/C++. Algoritmii prezentaţi în curs sunt descrişi în *limbaj natural* sau în *limbaj algoritmic* iar implementările sunt în limbajul de programare Java. Java este un limbaj orientat-obiect, dar noi vom utiliza foarte puţin această particularitate. Sunt prezentate toate elementele limbajului de programare Java necesare pentru acest curs *dar* ecesta *nu este* un curs de *programare în Java*.

Cunoştinţele minimale acceptate la sfârşitul cursului rezultă din *Legea nr. 288 din 24 iunie 2004 privind organizarea studiilor universitare* şi, de exemplu, din *Ghidul calităţii în învăţământul superior*[3]. Aici se precizează faptul că diploma de licenţă se acordă unui absolvent al programului de studii care: **demonstrează acumulare de cunoştinţe** şi **capacitatea de a înţelege** aspecte din domeniul de studii în care s-a format, **poate folosi** atât cunoştinţele acumulate precum şi capacitatea lui de înţelegere a fenomenelor printr-o **abordare profesională** în domeniul de activitate, **a acumulat competenţe** necesare **demonstrării**, **argumentării** şi **rezolvării problemelor** din domeniul de studii considerat, şi-a dezvoltat **deprinderi de învăţare** necesare procesului de educaţie continuă.

[2]Aprobată prin O.M:Ed.C. nr.5287/15.11.2004

[3]Editura Universităţii din Bucureşti, 2004; Capitolul 4, *Calitatea programelor de studii universitare*, Prof.univ.dr. Gabriela M. Atanasiu - Universitatea Tehnică "Gh.Asachi" din Iaşi

# Cuprins

# Capitolul 1

# Noţiuni fundamentale

În general, studenţii din anul I au cunoştinţe de programare în Pascal sau C/C++. Noi vom prezenta implementările algoritmilor în Java. Nu are prea mare importanţă dacă este Java, C/C++, Pascal sau alt limbaj de programare. Oricare ar fi limbajul de programare, trebuie să ştim în primul rând cum se reprezintă numerele în memoria calculatorului. Altfel putem avea surprize ciudate.

## 1.1 Programe ciudate

Dacă nu suntem atenţi la valorile pe care le pot lua variabilele cu care lucrăm, putem obţine rezultate greşite chiar dacă modalitatea de rezolvare a problemei este corectă. Prezentăm astfel de situaţii în Pascal, C/C++ şi Java.

### 1.1.1 Un program ciudat în Pascal

Iată un program Pascal în care dorim să calculăm suma $20.000 + 30.000$.

```
var x,y,z:integer;
BEGIN
  x:=20000;
  y:=30000;
  z:=x+y;
  write(x,'+',y,'=',z);
END.
```

Deşi ne aşteptam să apară ca rezultat 50.000, surpriza este că pe ecran apare

```
20000+30000=-15536
```

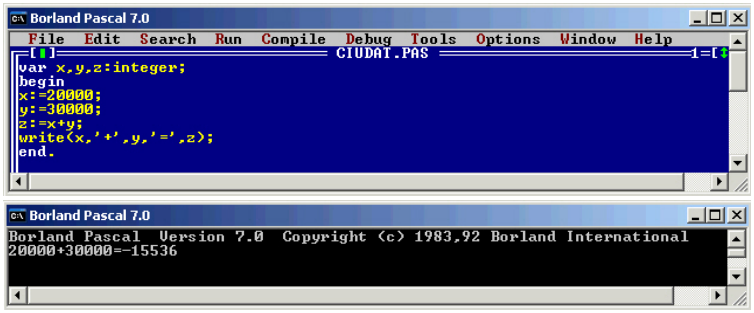



Figura 1.1: Un program ciudat în Pascal

### 1.1.2 Un program ciudat în C++

Iată un program în C++ în care dorim să calculăm suma $20.000 + 30.000$.

```
#include<iostream.h>
int main()
{
  int x,y,z;
  x=20000;  y=30000;  z=x+y;
  cout << x <<"+"<<y<<"="<<z;
  return 0;
}
```

Deşi ne aşteptam să apară ca rezultat 50.000, surpriza este că pe ecran apare

```
20000+30000=-15536
```

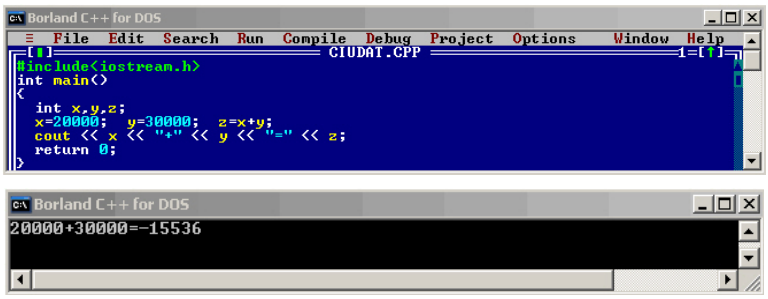



Figura 1.2: Un program ciudat în C++

### 1.1.3 Un program ciudat în Java

Iată un program în C++ în care dorim să calculăm suma $200.000 * 300.000$.

```
class Ciudat {
  public static void main(String args[]) {
    int x,y,z;
    x=200000;
    y=300000;
    z=x*y;
    System.out.println(x+"*"+y+"="+z);
  }
}
```

Deşi ne aşteptam să apară ca rezultat 60.000.000.000, surpriza este că pe ecran apare

```
200000*300000=-129542144
```

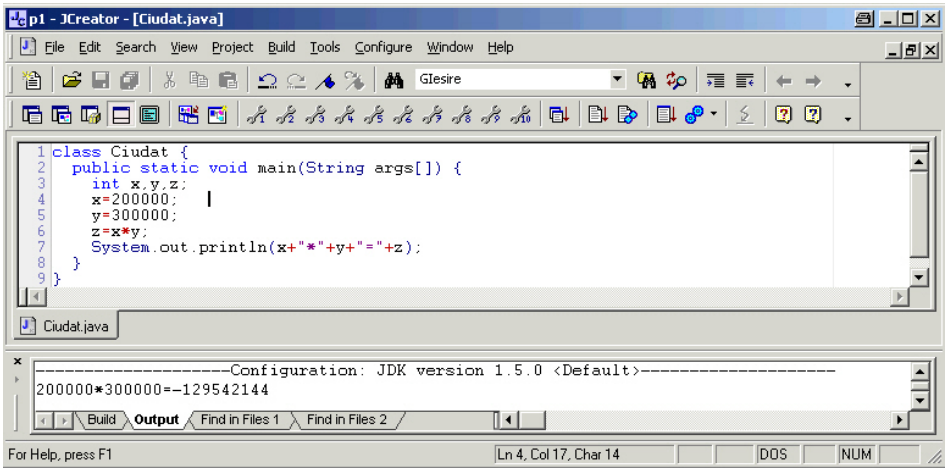



Figura 1.3: Un program ciudat în Java

Calculul cu numerele întregi este relativ simplu. Calculele sunt făcute într-o aritmetică modulo $N = 2^n$ unde $n$ este numărul de biţi ai cuvântului maşină. Există maşini pe 16, 32 şi 64 biţi pentru care $N$ este aproximativ egal cu $6 \times 10^4$, $4 \times 10^9$ şi respectiv $2 \times 10^{19}$.

Se pot reprezenta şi numerele întregi negative. Modul curent de reprezentare este în complement faţă de 2. În notaţie binară, bitul cel mai semnificativ este bitul de semn. Numerele negative sunt cuprinse între $-2^{n-1}$ şi $2^{n-1} - 1$.

Atunci când valorile obţinute din calcule depăşesc marginile permise de *tipul* variabilelor implicate în respectivele calcule, se pot obţine rezultate eronate.

### 1.1.4 Structura unui program Java

Un program simplu în Java are următoarea structură:

```
class numeClasa
{
        public static void main(String args[])
        {
                // declarări de variabile
                // instrucţiuni
        }
}
```

Programul prezentat în secţiunea anterioară se poate scrie sub forma:

```
class Ciudat
{
        public static void main(String args[])
        {
                // declarări de variabile
                int x,y,z;

                // instrucţiuni
                x=200000;
                y=300000;
                z=x*y;
                System.out.println(x+"*"+y+"="+z);
        }
}
```

**Clasa** este elementul de bază în Java. Cel mai simplu program în Java este format dintr-o clasă (numele clasei este la latitudinea programatorului; singura recomandare este să înceapă cu literă mare) şi funcţia **main**.

În exemplul de mai sus sunt declarate trei variabile (x, y şi z) de tip **int** (adică de *tip întreg cu semn*). Spaţiul alocat variabilelor de tip **int** este de 4 octeţi (32 biţi). Aceasta înseamnă că o astfel de variabilă poate avea valori între $-2^{63}$ şi $2^{63} - 1$. Valoarea maximă este de aproximativ 2 miliarde.

În programul anterior $x$ are valoarea 200.000 iar $y$ are valoarea 300.000, deci produsul are valoarea 60 miliarde care depăşeşte cu mult valoarea maximă de 2 miliarde.

În binar, 60 miliarde se scrie (folosint 36 biţi) sub forma

**1101**111110000100011101011000000000000

dar sunt reţinuţi numai 32 biţi din partea dreaptă, adică

11111000010001110101100000000000

Primul bit reprezintă bitul de semn (1 reprezintă semnul - iar 0 reprezintă semnul +). Această reprezentare trebuie gândită ca fiind o reprezentare în *cod*

*complementar* (ea este în *memoria calculatorului* şi toate numerele întregi cu semn sunt reprezentate în acest cod).

Reprezentarea în *cod direct* se obţine din reprezentarea în cod complementar (mai precis, trecând prin reprezentarea în *cod invers* şi adunând, în binar, 1):

**1**11110000100011101011000000000000 (cod complementar)
**1**00001111011100010100111111111111 (cod invers)
**1**00001111011100010101000000000000 (cod direct)

Din *codul direct* se obţine -129542144 în baza 10. Aceasta este explicaţia acelui *rezultat ciudat*!

## 1.2 Conversii ale datelor numerice

### 1.2.1 Conversia din baza 10 în baza 2

Fie $x = a_n...a_0$ numărul scris în baza 10. Conversia în baza 2 a numărului $x$ se efectuează după următoarele reguli:

- Se împarte numărul $x$ la 2 iar restul va reprezenta cifra de ordin 0 a numărului scris în noua bază ($b_0$).
- Câtul obţinut la împărţirea anterioară se împarte la 2 şi se obţine cifra de ordin imediat superior a numărului scris în noua bază. Secvenţa de împărţiri se repetă până când se ajunge la câtul 0.
- Restul de la a $k$-a împărţire va reprezenta cifra $b_{k-1}$. Restul de la ultima împărţire reprezintă cifra de ordin maxim în reprezentarea numărului în baza 2.

Metoda conduce la obţinerea rezultatului după un număr finit de împărtiri, întrucât în mod inevitabil se ajunge la un cât nul. În plus, toate resturile obţinute aparţin mulţimii $\{0, 1\}$.

**Exemplu**.

Fie $x = 13$ numărul în baza 10. Secvenţa de împărţiri este:

(1) se împarte 13 la 2 şi se obţine câtul 6 şi restul 1 (deci $b_0 = 1$)

(2) se împarte 6 la 2 şi se obţine câtul 3 şi restul 0 (deci $b_1 = 0$)

(3) se împarte 3 la 2 şi se obţine câtul 1 şi restul 1 (deci $b_2 = 1$)

(4) se împarte 1 la 2 şi se obţine câtul 0 şi restul 1 (deci $b_3 = 1$).

Prin urmare $(13)_{10} = (1101)_2$.

### 1.2.2 Conversia din baza 2 în baza 10

Dacă $y = b_n...b_1b_0$ este un număr în baza 2 , atunci reprezentarea în baza 10 se obţine efectuând calculul (în baza 10):

$$x = b_n 2^n + ... + b_1 2 + b_0.$$

**Exemplu**. Fie $y = 1100$. Atunci reprezentarea în baza 10 va fi

$$x = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 12.$$

### 1.2.3 Conversii între bazele 2 şi $2^r$

Pentru conversia unui număr din baza $p$ în baza $q$ se poate converti numărul din baza $p$ în baza 10, iar acesta se converteşte în baza $q$.

În cazul conversiei unui număr din baza $p = 2$ în baza $q = 2^r$ se poate evita trecerea prin baza 10 procedându-se în modul următor: se formează grupuri de câte $r$ cifre pornind de la ultima cifră din dreapta, înspre stânga. Fiecare grup de $r$ cifre va fi convertit într-o cifră a bazei $q$.

Fie, spre exemplu: $p = 2$, $q = 16 = p^4$ şi $x = (1011010)_2$.

Se obţin următoarele grupuri de câte 4 cifre binare:

$$(1010)_2 = A_{16} \text{ şi } (0101)_2 = 5_{16}.$$

Deci scrierea numărului $x$ în baza 16 este: $(5A)_{16}$.

Se observă că a fost completată cu 0, spre stânga, cea mai din stânga grupă, până la formarea grupei complete de 4 cifre binare.

În cazul conversiei unui număr din baza $p = 2^r$ în baza $q = 2$ se poate de asemenea evita trecerea prin baza 10 procedându-se în modul următor: fiecare cifră din reprezentarea în baza $p = 2^r$ se înlocuieşte cu $r$ cifre binare care reprezintă scrierea respectivei cifre în baza 2.

Fie, spre exemplu: $p = 16 = 2^4$, $q = 2$ şi $x = (3A)_{16}$.

Se fac următoarele înlocuiri de cifre:

$$3 \to 0011, \ A \to 1010.$$

Deci scrierea numărului $x$ în baza 2 este: $(111010)_2$.

Se observă că nu apar cifrele 0 din stânga *scrierii brute* $(00111010)_2$ obţinute prin înlocuiri.

# Capitolul 2

# Structuri de date

Înainte de a elabora un algoritm, trebuie să ne gândim la modul în care reprezentăm datele.

## 2.1 Date şi structuri de date

### 2.1.1 Date

Datele sunt entităţi purtătoare de informaţie. În informatică, o *dată* este un *model de reprezentare* a informaţiei, accesibil unui anumit *procesor* (om, unitate centrală, program), model cu care se poate opera pentru a obţine noi informaţii despre fenomenele, procesele şi obiectele lumii reale. În functie de modul lor de organizare, datele pot fi: *elementare* (simple) sau structurate.

*Datele elementare* au caracter atomic, în sensul că nu pot fi descompuse în alte date mai simple. Astfel de date sunt cele care iau ca valori *numere* sau *şiruri de caractere*. O *dată elementară* apare ca o entitate indivizibilă atât din punct de vedere al informaţiei pe care o reprezintă cât şi din punct de vedere al procesorului care o prelucrează.

O *dată elementară* poate fi privită la *nivel logic* (la nivelul procesorului uman) sau la nivel *fizic* (la nivelul calculatorului).

Din punct de vedere *logic*, o *dată* poate fi definită ca un triplet de forma

$$(identificator, atribute, valori).$$

Din punct de vedere *fizic*, o *dată* poate fi definită ca o *zonă de memorie* de o anumită *lungime*, situată la o anumită *adresă* absolută, în care sunt *memorate* în timp şi într-o formă specifică *valorile* datei.

*Identificatorul* este un *simbol* asociat datei pentru a o distinge de alte date şi pentru a o putea referi în cadrul programului.

*Atributele* sunt *proprietăţii* ale datei şi precizează modul în care aceasta va fi tratată în cadrul procesului de prelucrare. Dintre atribute, cel mai important este atributul de *tip* care defineştete apartenenţa datei la o anumită *clasă de date*.

O *clasă de date* este definită de *natura* şi *domeniul valorilor* datelor care fac parte din clasa respectivă, de *operaţiile* specifice care se pot efectua asupra datelor şi de modelul de *reprezentare internă* a datelor. Astfel, există date de tip *întreg*, de tip *real*, de tip *logic*, de tip *şir de caractere*, etc.

O mulţime de date care au aceleaşi caracteristici se numeşte *tip de date*. Evident, un *tip de date* este o *clasă de date* cu acelaşi mod de interpretare logică şi reprezentare fizică şi se caracterizează prin *valorile* pe care le pot lua datele şi prin *operaţiile* care pot fi efectuate cu datele de tipul respectiv.

De exemplu, *tipul întreg* se caracterizează prin faptul că datele care îi aparţin pot lua doar valori întregi, şi asupra lor pot fi efectuate operaţii aritmetice clasice (adunare, scădere, înmulţire, împărţire în mulţimea numerelor întregi, comparaţii).

Se poate considera că datele organizate sub forma tablourilor unidimensionale formează *tipul vector* iar datele organizate sub forma tablourilor bidimensionale formează *tipul matrice*.

În funcţie de natura elementelor care o compun, o structură de date poate fi:

- *omogenă*, atunci când toate elementele au *acelaşi tip*;
- *neomogenă*, atunci când elementele componente au *tipuri diferite*.

În functie de numărul datelor care o compun, o structură de date poate fi:

- *statică*, atunci când numărul de componente este fixat;
- *dinamică*, atunci când numărul de componente este variabil.

Din punct de vedere al modului în care sunt utilizate datele pot fi:

- *Constante*. Valoarea lor nu este şi nu poate fi modificată în cadrul algoritmului, fiind fixată de la începutul acestuia. O *constantă* este o dată care păstrează aceeaşi valoare pe tot parcursul procesului de prelucrare. Pentru constantele care nu au nume, însăşi valoarea lor este cea prin care se identifică. Constante care au nume (identificator) sunt iniţializate cu o valoare în momentul declarării.

- *Variabile*. Valoarea lor poate fi modificată în cadrul algoritmului. În momentrul declarării lor, variabilele pot fi *iniţializate* (li se atribuie o valoare) sau pot fi *neiniţializate* (nu li se atribuie nici o valoare). O *variabilă* este o dată care nu păstrează neapărat aceeaşi valoare pe parcursul procesului de prelucrare.

Tipul unei date trebuie să fie precizat, în cadrul programului de prelucrare, printr-o *declaraţie de tip* ce precede utilizarea respectivei constante sau variabile.

*Valorile* datei pot fi numere, sau valori de adevăr, sau şiruri de caractere, etc.

### 2.1.2 Structuri de date

Datele apar frecvent sub forma unor colecţii de date de diferite tipuri, menite să faciliteze prelucrarea în cadrul rezolvării unei anumite probleme concrete.

*Datele structurate*, numite uneori şi *structuri de date*, sunt constituite din mai multe date elementare (uneori de acelaşi tip, alteori de tipuri diferite), grupate cu un anumit scop şi după anumite reguli.

**Exemple**.

1. Un şir finit de numere reale $a_1, a_2, ..., a_n$ poate fi reprezentat ca o dată structurată (*tablou unidimensional* sau *vector*).

2. O matrice

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & \cdots & \ddots & \cdots \\ a_{m,1} & a_{m,1} & \cdots & a_{m,n} \end{bmatrix}$$

poate fi reprezentată ca o dată structurată (*tablou bidimensional*) specificînd fiecare element prin doi indici (de linie şi de coloană).

O *structură de date* este deci o colecţie de date, eventual de tipuri diferite, pe care s-a definit o anumită organizare şi căreia îi este specific un anumit mod de identificare a elementelor componente. Componetele unei structuri de date pot fi identificate prin *nume* sau prin *ordinea* pe care o ocupă în cadrul structurii.

Dacă accesul la o anumită componentă a structurii de date se poate face fără să ţinem seama de celelalte componente, vom spune că structura de date este cu *acces direct*. În schimb, dacă accesul la o componentă a structurii de date se poate face numai ţinând cont de alte câmpuri ale structurii (în conformitate cu ordinea structurii, printr-un proces de traversare) atunci vom spune că structura este cu *acces secvenţial*.

Structurile de date pot fi create pentru a fi depozitate în *memoria internă* (aceste structuri de date se numesc *structuri interne*) sau în *memoria externă* (se numesc *structuri externe*, sau *fişiere*). Structurile *interne* au un caracter de *date temporare* (ele dispar odată cu încetarea activităţii de prelucrare) iar cele *externe* au un caracter de *date permanente* (mai bine spus, de lungă durată).

Dacă pe lângă componentele structurii se înregistrează pe suport şi alte date suplimentare care să materializeze relaţia de ordonare, atunci structura de date respectivă este *explicită*, în caz contrar este *implicită*. De exemplu, structura de date de tip *tablou* este o structură *implicită* de date iar structura de date de tip *listă liniară* este o structură *explicită* de date.

Asupra structurilor de date se pot efectua operaţii care se referă structura respectivă sau la valorile datelor componente. Cele mai importante operaţii sunt:

– operaţia de *creare*, care constă în memorarea pe suportul de memorie a structurii de date în forma sa iniţială,

– operaţia de *consultare*, care constă în accesul la elementele structurii în vederea prelucrării valorilor acestora, şi

– operaţia de *actualizare*, care constă în *adăugarea* de noi elemente, sau *eliminarea* elementelor care nu mai sunt necesare, sau *modificarea* valorilor unor componente ale structurii.

Toate structurile de date la fel *organizate* şi pe care s-au definit aceleaşi *operaţii*, poartă numele de *tip de structură de date*. Dacă analizăm însă operaţiile care se efectuează asupra unei structuri de date, vom putea vedea că toate acestea se reduc la executarea, eventual repetată, a unui grup de operaţii specifice numite *operaţii de bază*.

## 2.2 Structuri şi tipuri de date abstracte

### 2.2.1 Structuri de date abstracte

*Abstractizarea datelor* reprezintă de fapt concentrarea asupra *esenţialului*, ignorând detaliile (sau altfel spus, contează "ce" nu "cum").

Stăpânirea aplicaţiilor complexe se obţine prin *descompunerea în module*.

Un *modul* trebuie să fie simplu, cu complexitatea ascunsă în interiorul lui, şi să aibă o interfaţă simplă care să permită folosirea lui fără a cunoaşte implementarea.

O *structură de date abstractă* este un *modul* constând din *date* şi *operaţii*. Datele sunt *ascunse* în interiorul modulului şi pot fi accesate prin intermediul operaţiilor. Structura de date este *abstractă* deoarece este cunoscută numai *interfaţa* structurii, nu şi *implementarea* (operaţiile sunt date explicit, valorile sunt definite implicit, prin intermediul operaţiilor).

### 2.2.2 Tipuri de date abstracte

Procesul de *abstractizare* se referă la două aspecte:

- *abstractizarea procedurală*, care separă proprietăţile logice ale unei *acţiuni* de detaliile implementării acesteia
- *abstractizarea datelor*, care separă proprietăţile logice ale *datelor* de detaliile reprezentării lor

O *structură de date abstracte* are un singur *exemplar* (o singură *instanţă*). Pentru a crea mai multe *exemplare* ale structurii de date abstracte se defineşte un *tip de date abstract*. În Java, de exemplu, *clasa* asigură un mod direct de definire a oricărui *tip de date abstract*.

## 2.3 Structuri de date elementare

### 2.3.1 Liste

O *listă* este o *colecţie de elemente* de informaţie (noduri) *aranjate* într-o anumită ordine. *Lungimea* unei liste este numărul de noduri din listă. Structura corespunzatoare de date trebuie să ne permită să determinăm eficient care este *primul/ultimul* nod în structură şi care este *predecesorul/succesorul* unui nod dat (dacă există). Iată cum arată cea mai simplă listă, *lista liniară*:

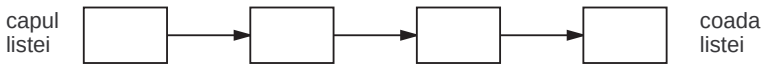



Figura 2.1: Listă liniară

O *listă circulară* este o listă în care, după *ultimul nod*, urmează *primul nod*, deci fiecare nod are *succesor* şi *predecesor*.

Câteva dintre *operaţiile* care se efectuează asupra listelor sunt: *inserarea* (adăugarea) unui nod, *extragerea* (ştergerea) unui nod, *concatenarea* unor liste, numărarea elementelor unei liste etc.

Implementarea unei liste se realizează în două moduri: *secvenţial* şi *înănţuit*.

*Implementarea secvențială* se caracterizează prin plasarea nodurilor în locaţii succesive de memorie, în conformitate cu ordinea lor în listă. Avantajele acestui mod de implementare sunt *accesul* rapid la predecesorul/succesorul unui nod şi *găsirea* rapidă a primului/ultimului nod. Dezavantajele sunt modalităţile relativ complicate de *inserarea/ştergere* a unui nod şi faptul că, în general, nu se foloseşte întreaga memorie alocată listei.

*Implementarea înlănţuită* se caracterizează prin faptul că fiecare nod conţine două părţi: *informaţia* propriu-zisă şi *adresa* nodului succesor. Alocarea memoriei pentru fiecare nod se poate face în mod dinamic, în timpul rulării programului. *Accesul* la un nod necesită *parcurgerea* tuturor predecesorilor săi, ceea ce conduce la un consum mai mare de timp pentru această operaţie. În schimb, operaţiile de *inserare/ştergere* sunt foarte rapide. Se consumă exact atât spaţiu de memorie cât este necesar dar, evident, apare un consum suplimentar de memorie pentru înregistrarea legăturii către nodul succesor. Se pot folosi două adrese în loc de una, astfel încât un nod să conţină pe langă adresa nodului succesor şi adresa nodului predecesor. Obţinem astfel o *listă dublu inlănţuită*, care poate fi traversată în ambele direcţii.

Listele înlănţuite pot fi reprezentate prin tablouri. În acest caz, adresele nodurilor sunt de fapt indici ai tabloului.

O alternativă este să folosim două tablouri *val* şi *next* astfel: să memorăm informaţia fiecarui nod $i$ în locaţia $val[i]$, iar adresa nodului său succesor în locaţia $next[i]$. Indicele locaţiei primului nod este memorat în variabila $p$. Vom conveni ca, pentru cazul listei vide, să avem $p = 0$ şi $next[u] = 0$ unde $u$ reprezintă ultimul nod din listă. Atunci, $val[p]$ va conţine informaţia primului nod al listei, $next[p]$ adresa celui de-al doilea nod, $val[next[p]]$ informaţia din al doilea nod, $next[next[p]]$ adresa celui de-al treilea nod, etc. Acest mod de reprezentare este simplu dar apare problema gestionării locaţiilor libere. O soluţie este să reprezentăm locaţiile libere tot sub forma unei liste înlanţuite. Atunci, ştergerea unui nod din lista iniţială implică inserarea sa în lista cu locaţii libere, iar inserarea unui nod în lista iniţială implică ştergerea sa din lista cu locaţii libere. Pentru implementarea listei de locaţii libere, putem folosi aceleaşi tablouri dar avem nevoie de o altă variabilă, $freehead$, care să conţină indicele primei locaţii libere din *val* şi *next*. Folosim aceleaşi convenţii: dacă $freehead = 0$ înseamnă că nu mai avem locaţii libere, iar $next[ul] = 0$ unde $ul$ reprezintă ultima locaţie liberă.

Vom descrie in continuare două tipuri de liste particulare foarte des folosite.

### 2.3.2 Stive şi cozi

O *stivă* este o listă liniară cu proprietatea că operaţiile de inserare/extragere a nodurilor se fac în/din coada listei. Dacă nodurile A, B, C sunt inserate într-o stivă în această ordine, atunci primul nod care poate fi şters/extras este C. În mod echivalent, spunem că ultimul nod inserat este singurul care poate fi şters/extras. Din acest motiv, stivele se mai numesc şi *liste LIFO* (**L**ast **I**n **F**irst **O**ut).

Cel mai natural mod de reprezentare pentru o stivă este implementarea secvenţială într-un tablou $S[1..n]$, unde $n$ este numărul maxim de noduri. Primul nod va fi memorat în $S[1]$, al doilea în $S[2]$, iar ultimul în $S[top]$, unde *top* este o variabilă care conţine adresa (indicele) ultimului nod inserat. Iniţial, când stiva este vidă, avem (prin convenţie) $top = 0$.

O *coadă* este o listă liniară în care inserările se fac doar în capul listei, iar ştergerile/extragerile se fac doar din coada listei. Din acest motiv, cozile se mai numesc şi *liste FIFO* (**F**irst **I**n **F**irst **O**ut).

O reprezentare secvenţială pentru o coadă se obţine prin utilizarea unui tablou $C[0..n-1]$, pe care îl tratăm ca şi cum ar fi circular: după locaţia $C[n-1]$ urmează locaţia $C[0]$. Fie *tail* variabila care conţine indicele locaţiei predecesoare primei locaţii ocupate şi fie *head* variabila care conţine indicele locaţiei ocupate ultima oară. Variabilele *head* şi *tail* au aceeaşi valoare atunci şi numai atunci când coada este vidă. Iniţial, avem $head = tail = 0$.

Trebuie să observăm faptul că testul de coadă vidă este acelaşi cu testul de coadă plină. Dacă am folosi toate cele $n$ locaţii la un moment dat, atunci nu am putea distinge între situaţia de "coadă plină" şi cea de "coadă vidă", deoarece în ambele situaţii am avea $head = tail$. În consecinţă, vom folosi efectiv, în orice moment, cel mult $n-1$ locaţii din cele $n$ ale tabloului $C$.

### 2.3.3 Grafuri

Un *graf* este o pereche $G =< V, M >$, unde $V$ este o mulţime de *vârfuri*, iar $M \subseteq V \times V$ este o mulţime de *muchii*. O *muchie* de la vârful $a$ la vârful $b$ este notată cu perechea ordonată $(a, b)$, dacă graful este *orientat*, şi cu multimea $\{a, b\}$, dacă graful este *neorientat*.

Două vârfuri unite printr-o muchie se numesc *adiacente*. Un vârf care este extremitatea unei singure muchii se numeşte *vârf terminal*.

Un *drum* este o succesiune de muchii de forma

$$(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n)$$

sau de forma

$$\{a_1, a_2\}, \{a_2, a_3\}, ..., \{a_{n-1}, a_n\}$$

după cum graful este *orientat* sau *neorientat*. *Lungimea drumului* este egală cu numărul muchiilor care îl constituie. Un *drum simplu* este un drum în care nici un vârf nu se repetă. Un *ciclu* este un drum care *este simplu*, cu excepţia primului şi ultimului vârf, care coincid. Un *graf aciclic* este un graf fără cicluri.

Un graf neorientat este *conex*, dacă între oricare două vârfuri există un *drum*. Pentru grafuri orientate, această notiune este întărită: un graf orientat este *tare conex*, dacă între oricare două vârfuri $i$ şi $j$ există un *drum* de la $i$ la $j$ şi un *drum* de la $j$ la $i$.

Vârfurilor unui graf li se pot ataşa *informaţii* (numite *valori*), iar muchiilor li se pot ataşa *informaţii* numite uneori *lungimi* sau *costuri*.

Există cel puţin trei moduri de *reprezentare* ale unui graf:

• Printr-o *matrice de adiacenţă* $A$, în care $A[i, j] = true$ dacă vârfurile $i$ şi $j$ sunt *adiacente*, iar $A[i, j] = false$ în caz contrar. O altă variantă este să-i dăm lui $A[i, j]$ valoarea lungimii muchiei dintre vârfurile $i$ şi $j$, considerand $A[i, j] = +\infty$ atunci când cele două vârfuri nu sunt adiacente. Cu această reprezentare, putem verifica uşor dacă două vârfuri sunt adiacente. Pe de altă parte, dacă dorim să aflăm toate vârfurile adiacente unui vârf dat, trebuie să analizăm o întreagă linie din matrice. Aceasta necesită $n$ operaţii (unde $n$ este numărul de vârfuri în graf), independent de numărul de muchii care conectează vârful respectiv.

• Prin *liste de adiacenţă*, adică prin ataşarea la fiecare vârf $i$ a listei de vârfuri *adiacente* (pentru grafuri orientate, este necesar ca muchia să plece din $i$). Într-un graf cu $m$ muchii, suma lungimilor listelor de adiacentă este $2m$, dacă graful este *neorientat*, respectiv $m$, dacă graful este *orientat*. Dacă numărul muchiilor în graf este mic, această reprezentare este preferabilă din punct de vedere al memoriei necesare. Totuşi, pentru a determina dacă două vârfuri $i$ şi $j$ sunt adiacente, trebuie să analizăm lista de adiacentă a lui $i$ (şi, posibil, lista de adiacentă a lui $j$), ceea ce este mai puţin eficient decât consultarea unei valori logice în matricea de adiacenţă.

• Printr-o *listă de muchii*. Această reprezentare este eficientă atunci când avem de examinat toate muchiile grafului.

### 2.3.4 Arbori binari

Un *arbore* este un graf neorientat, aciclic şi conex. Sau, echivalent, un arbore este un graf neorientat în care există exact un drum între oricare două vârfuri.

Un arbore reprezentat pe niveluri se numeşte *arbore cu rădăcină*. Vârful plasat pe nivelul 0 se numeşte *rădăcina arborelui*. Pe fiecare nivel $i > 0$ sunt plasate vârfurile pentru care lungimea drumurilor care le leagă de rădăcină este $i$.

Vârfurile de pe un nivel $i > 0$ legate de acelaşi vârf $j$ de pe nivelul $i - 1$ se numesc *descendenţii direcţi (fiii)* vârfului $j$ iar vârful $j$ se numeşte *ascendent direct (tată)* al acestor vârfuri.

Dacă există un drum de la un vârf $i$ de pe nivelul $ni$ la un vârf $j$ de pe nivelul $nj > ni$, atunci vârful $i$ se numeşte *ascendent* al lui $j$, iar vârful $j$ se numeşte *descendent* al lui $i$.

Un *vârf terminal* (sau *frunză*) este un vârf fără descendenţi. Vârfurile care nu sunt terminale se numesc *neterminale*.

Un arbore în care orice vârf are cel mult doi descendenţi se numeşte *arbore binar*.

Într-un arbore cu rădăcină (reprezentat pe niveluri), *adâncimea* unui vârf este lungimea drumului dintre rădăcină şi acest vârf iar *înălţimea* unui vârf este lungimea celui mai lung drum dintre acest vârf şi un vârf terminal.

*Înălţimea arborelui* este înălţimea rădăcinii.

Într-un arbore binar, numărul maxim de vârfuri aflate pe nivelul $k$ este $2^k$. Un arbore binar de înălţime $k$ are cel mult $2^{k+1} - 1$ vârfuri, iar dacă are exact $2^{k+1} - 1$ vârfuri, se numeşte *arbore plin*.

Varfurile unui arbore plin se numeroteaza în ordinea nivelurilor. Pentru acelaşi nivel, numerotarea se face în arbore de la stânga la dreapta.

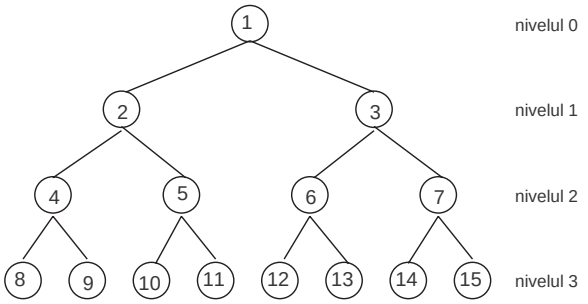



Figura 2.2: Arbore binar plin

Un arbore binar cu $n$ vârfuri şi de înălţime $k$ este *complet*, dacă se obţine din arborele binar plin de înălţime $k$, prin eliminarea, dacă este cazul, a vârfurilor numerotate cu $n + 1$, $n + 2$, ..., $2^{k+1} - 1$.

Acest tip de arbore se poate reprezenta secvenţial folosind un tablou $T$, punând vârfurile de adâncime $k$, de la stânga la dreapta, în poziţiile $T[2^k]$, $T[2^k+1]$, ..., $T[2^{k+1}-1]$ (cu posibila excepţie a ultimului nivel care poate fi incomplet).

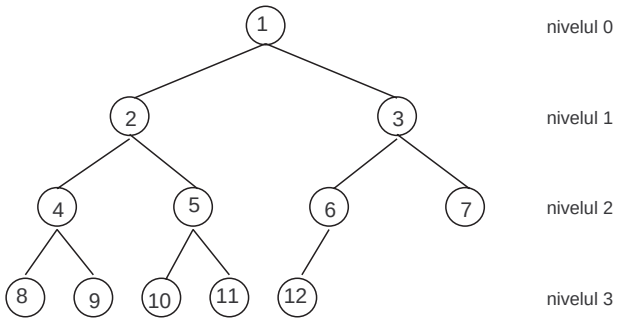



Figura 2.3: Arbore binar complet

Tatăl unui vârf reprezentat în $T[i]$, $i > 0$, se află în $T[i/2]$. Fiii unui vârf reprezentat în $T[i]$ se află, dacă există, în $T[2i]$ şi $T[2i+1]$.

### 2.3.5 Heap-uri

Un *max-heap* (heap="gramadă ordonată", în traducere aproximativă) este un arbore binar complet, cu următoarea proprietate: valoarea fiecarui vârf este mai mare sau egală cu valoarea fiecărui fiu al său.

Un *min-heap* este un arbore binar complet în care valoarea fiecarui vârf este mai mică sau egală cu valoarea fiecărui fiu al său.

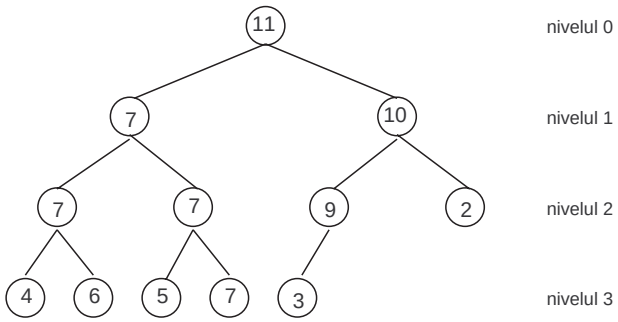



Figura 2.4: Max-heap

Acelaşi heap poate fi reprezentat secvenţial prin următorul tablou:

| 11 | 7 | 10 | 7 | 7 | 9 | 2 | 4 | 6 | 5 | 7 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Caracteristica de bază a acestei structuri de date este că modificarea valorii unui vârf se face foarte eficient, păstrându-se proprietatea de heap.

De exemplu, într-un max-heap, dacă valoarea unui vârf creşte, astfel încât depăşeşte valoarea tatălui, este suficient să schimbăm între ele aceste două valori şi să continuăm procedeul în mod ascendent, până când proprietatea de heap este restabilită. Dacă, dimpotrivă, valoarea vârfului scade, astfel încât devine mai mică decât valoarea cel puţin a unui fiu, este suficient să schimbăm intre ele valoarea modificată cu cea mai mare valoare a fiiilor, apoi să continuăm procesul în mod descendent, până când proprietatea de heap este restabilită.

Heap-ul este structura de date ideală pentru extragerea maximului/minimului dintr-o mulţime, pentru inserarea unui vârf, pentru modificarea valorii unui vârf. Sunt exact operaţiile de care avem nevoie pentru a implementa o *listă dinamică de priorităţi*: valoarea unui vârf va da prioritatea evenimentului corespunzator.

Evenimentul cu prioritatea cea mai mare/mică se va afla mereu la radacina heap-ului, iar prioritatea unui eveniment poate fi modificată în mod dinamic.

### 2.3.6 Structuri de mulţimi disjuncte

Să presupunem că avem $N$ elemente, numerotate de la 1 la $N$. Numerele care identifică elementele pot fi, de exemplu, indici intr-un tablou unde sunt memorate valorile elementelor. Fie o partitie a acestor $N$ elemente, formată din submulţimi două câte două disjuncte: $S1$, $S2$, ... . Presupunem că ne interesează reuniunea a două submulţimi, $S_i \cup S_j$.

Deoarece submulţimile sunt două câte două disjuncte, putem alege ca etichetă pentru o submulţime oricare element al ei. Vom conveni ca elementul minim al unei mulţimi să fie eticheta mulţimii respective. Astfel, multimea $\{3, 5, 2, 8\}$ va fi numită "multimea 2".

Vom aloca tabloul $set[1..N]$, în care fiecărei locatii $set[i]$ i se atribuie eticheta submulţimii care conţine elementul $i$. Avem atunci proprietatea: $set[i] \leq i$, pentru $1 \leq i \leq N$. Reuniunea submulţimilor etichetate cu $a$ şi $b$ se poate realiza astfel:

**procedure** $reuniune(a, b)$
  $i \leftarrow a$;
  $j \leftarrow b$
  **if** $i > j$
    **then** interschimbă $i$ şi $j$
  **for** $k \leftarrow j$ **to** $N$ **do**
    **if** $set[k] = j$
      **then** $set[k] \leftarrow i$

# Capitolul 3

# Algoritmi

## 3.1 Etape în rezolvarea problemelor

Principalele etape care se parcurg în rezolvarea unei probleme sunt:

(a) Stabilirea *datelor* iniţiale şi a *obiectivului* (ce trebuie determinat).

(b) Alegerea *metodei* de rezolvare.

(c) *Aplicarea* metodei pentru date concrete.

**Exemplu.**
Să presupunem că problema este rezolvarea, în $\mathbb{R}$, a ecuaţiei $x^2 - 3x + 2 = 0$.

(a) *Datele* iniţiale sunt reprezentate de către coeficienţii ecuaţiei iar obiectivul este determinarea rădăcinilor reale ale ecuaţiei.

(b) Vom folosi *metoda* de rezolvare a ecuaţiei de gradul al doilea având forma generală $ax^2 + bx + c = 0$. Această metodă poate fi *descrisă* astfel:

Pasul 1. Se calculează discriminantul: $\Delta = b^2 - 4ac$.

Pasul 2. **Dacă** $\Delta > 0$

**atunci** ecuaţia are două rădăcini reale distincte: $x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$

**altfel**, **dacă** $\Delta = 0$

**atunci** ecuaţia are o rădăcina reală dublă: $x_{1,2} = \frac{-b}{2a}$

**altfel** ecuaţia nu are rădăcini reale.

(c) *Aplicarea* metodei pentru datele problemei ($a = 1, b = -3, c = 2$) conduce la rezultatul: $x_1 = 1$, $x_2 = 2$.

## 3.2 Algoritmi

### 3.2.1 Ce este un algoritm?

> Un **algoritm** este o *succesiune de operaţii* aritmetice şi/sau logice care, aplicate asupra unor *date*, permit obţinerea rezultatului unei probleme din clasa celor pentru care a fost conceput.

Să observăm că nu apare în definiţie cuvântul "calculator"; algoritmii nu au neapărat legătură cu calculatorul. Totuşi, în acest curs ne vom concentra aproape exclusiv pe algoritmi care pot fi implementaţi rezonabil pe calculator. Altfel spus, fiecare pas din algoritm trebuie astfel gândit încât ori este suportat direct de către limbajul de programare favorit (operaţii aritmetice, cicluri, recursivitate, etc) ori este asemănător cu ceva învăţat mai înainte (sortare, căutare binară, parcurgere în adâncime, etc).

Secvenţa de paşi prin care este descrisă metoda de rezolvare a ecuaţiei de gradul al doilea (prezentată în secţiunea anterioară) este un exemplu de algoritm. Calculul efectuat la Pasul 1 este un exemplu de operaţie aritmetică, iar analiza semnului discriminantului (Pasul 2) este un exemplu de operaţie logică.

*Descrierea* unui algoritm presupune *precizarea datelor* iniţiale şi *descrierea prelucrărilor* efectuate asupra acestora. Astfel, se poate spune că:

$$\textbf{algoritm} = \textbf{date} + \textbf{prelucrări}$$

Al-Khwarizmi a fost cel care a folosit pentru prima dată *reguli precise şi clare* pentru a *descrie* procese de calcul (operaţii aritmetice fundamentale) în lucrarea sa "Scurtă carte despre calcul algebric". Mai târziu, această *descriere* apare sub denumirea de *algoritm* în "Elementele lui Euclid". *Algoritmul lui Euclid* pentru calculul celui mai mare divizor comun a două numere naturale este, se pare, primul *algoritm* cunoscut în matematică.

În matematică noţiunea de *algoritm* a primit mai multe definiţii: algoritmul normal al lui A. A. Markov, algoritmul operaţional al lui A. A. Leapunov, maşina Turing, funcţii recursive, sisteme POST. S-a demonstrat că aceste definiţii sunt echivalente din punct de vedere matematic.

În informatică există de asemenea mai multe definiţii pentru noţiunea de *algoritm*. De exemplu, în [35] noţiunea de algoritm se defineşte astfel:

Un *algoritm* este sistemul virtual

$$A = (M, V, P, R, Di, De, Mi, Me)$$

constituit din următoarele elemente:

> $M$ - *memorie internă* formată din *locaţii de memorie* şi utilizată pentru stocarea temporară a valorilor variabilelor;

$V$ - mulţime de *variabile* definite în conformitate cu *raţionamentul* $R$, care utilizează memoria $M$ pentru stocarea valorilor din $V$;

$P$ - *proces de calcul* reprezentat de o colecţie de instrucţiuni/comenzi exprimate într-un limbaj de reprezentare (de exemplu, limbajul pseudocod); folosind memoria virtuală $M$ şi mulţimea de variabile $V$, instrucţiunile implementează/codifică tehnicile şi metodele care constituie *raţionamentul* $R$; execuţia instrucţiunilor procesului de calcul determină o dinamică a valorilor variabilelor; după execuţia tuturor instrucţiunilor din $P$, soluţia problemei se află în anumite locaţii de memorie corespunzătoare datelelor de ieşire $De$;

$R$ - *raţionament* de rezolvare exprimat prin diverse tehnici şi metode specifice domeniului din care face parte clasa de probleme supuse rezolvării (matematică, fizică, chimie etc.), care îmbinate cu tehnici de programare corespunzătoare realizează acţiuni/procese logice, utilizând memoria virtuală $M$ şi mulţimea de variabile $V$;

$Di$ - *date de intrare* care reprezintă valori ale unor parametri care caracterizează ipotezele de lucru/stările iniţiale ale problemei şi care sunt stocate în memoria $M$ prin intermediul instrucţiunilor de citire/intrare care utilizează mediul de intrare $Mi$;

$De$ - *date de ieşire* care reprezintă valori ale unor parametri care caracterizează soluţia problemei/stările finale; valorile datelor de ieşire sunt obţinute din valorile unor variabile generate de execuţia instrucţiunilor din procesul de calcul $P$, sunt stocate în memoria $M$, şi înregistrate pe un suport virtual prin intermediul instrucţiunilor de scriere/ieşire care utilizează mediul de ieşire $Me$; ;

$Mi$ - *mediu de intrare* care este un dispozitiv virtual de intrare/citire pentru preluarea valorilor datelor de intrare şi stocarea acestora în memoria virtuală $M$;

$Me$ - mediu de ieşire care este un dispozitiv virtual de ieşire/scriere pentru preluarea datelor din memoria virtuală $M$ şi înregistrarea acestora pe un suport virtual (ecran, hârtie, disc magnetic, etc.).

Un *limbaj* este un mijloc de transmitere a informaţiei.

Există mai multe tipuri de limbaje: *limbaje naturale* (engleză, română, etc), *limbaje ştiinţifice* (de exemplu limbajul matematic), limbaje algoritmice, limbaje de programare (de exemplu Pascal, C, Java), etc.

Un **limbaj de programare** este un limbaj artificial, riguros întocmit, care permite *descrierea algoritmilor* astfel încât să poată fi transmişi calculatorului cu scopul ca acesta să efectueze operaţiile specificate.

Un **program** este un *algoritm* tradus într-un *limbaj de programare*.

### 3.2.2 Proprietăţile algoritmilor

Principalele proprietăţi pe care trebuie să le aibă un algoritm sunt:

• *Generalitate*. Un algoritm trebuie să poată fi utilizat pentru o *clasă* întreagă *de probleme*, nu numai pentru o problemă particulară. Din această cauză, o metodă de rezolvare a unei ecuaţii particulare nu poate fi considerată *algoritm*.

• *Finitudine*. Orice algoritm trebuie să permită obţinerea rezultatului după un *număr finit* de prelucrări (paşi). Din această cauză, o metodă care nu asigură obţinerea rezultatului după un număr finit de paşi nu poate fi considerată *algoritm*.

• *Determinism*. Un algoritm trebuie să prevadă, fără ambiguităţi şi fără neclarităţi, modul de soluţionare a tuturor situaţiilor care pot să apară în rezolvarea problemei. Dacă în cadrul algoritmului nu intervin elemente aleatoare, atunci ori de câte ori se aplică algoritmul aceluiaşi set de date de intrare trebuie să se obţină acelaşi rezultat.

### 3.2.3 Tipuri de prelucrări

Prelucrările care intervin într-un algoritm pot fi *simple* sau *structurate*.

• *Prelucrările simple* sunt *atribuiri* de valori variabilelor, eventual prin evaluarea unor expresii;

• *Prelucrările structurate* pot fi de unul dintre tipurile:

– *Liniare*. Sunt secvenţe de prelucrări simple sau structurate care sunt efectuate în ordinea în care sunt specificate;

– *Alternative*. Sunt prelucrări caracterizate prin faptul că în funcţie de realizarea sau nerealizarea unei condiţii se alege una din două sau mai multe variante de prelucrare;

– *Repetitive*. Sunt prelucrări caracterizate prin faptul că aceeaşi prelucrare (simplă sau structurată) este repetată cât timp este îndeplinită o anumită condiţie.

## 3.3 Descrierea algoritmilor

*Algoritmii* nu sunt *programe*, deci ei nu trebuie specificaţi într-un limbaj de programare. Detaliile sintactice, de exemplu din Pascal, C/C++ sau Java, nu au nici o importanţă în elaborarea/proiectarea algoritmilor.

Pe de altă parte, descrierea în limba română (ca şi în limba engleză [15]) în mod uzual nu este o idee mai bună. Algoritmii au o serie de structuri - în special

condiţionale, repetitive, şi recursivitatea - care sunt departe de a putea fi *descrise* prea uşor în *limbaj natural*. La fel ca orice limbă vorbită, limba româna este plină de ambiguităţi, subînţelesuri şi nuanţe de semnificaţie, iar algoritmii trebuie să fie descrişi cu o acurateţe maxim posibilă.

Cea mai bună metodă de a *descrie* un *algoritm* este utilizarea *limbajului pseudocod*. Acesta foloseşte structuri ale limbajelor de programare şi matematicii pentru a *descompune algoritmul* în *paşi elementari* (propoziţii simple), dar care pot fi scrise folosind matematica, româna curată, sau un amestec al celor două.

Modul exact de structurare a pseudocodului este o alegere personală.

O descriere foarte bună a algoritmului *arată* structura internă a acestuia, *ascunde* detaliile care nu sunt semnificative, şi poate fi *implementată uşor* de către orice programator *competent* în orice limbaj de programare, chiar dacă el nu înţelege ce face acel algoritm. Un *pseudocod* bun, la fel ca şi un *cod* bun, face *algoritmul* mult mai uşor de înţeles şi analizat; el permite de asemenea, mult mai uşor, descoperirea greşelilor.

Pe de altă parte, proba clară se poate face numai pe baza unui program care să dea rezultatele corecte! Oamenii sunt oameni! Cineva poate să insiste că algoritmul lui este bun deşi ... nu este! Şi atunci ... programăm!

### 3.3.1 Limbaj natural

**Exemple**.

1. *Algoritmul lui Euclid*. Permite determinarea celui mai mare divizor comun (cmmdc) a două numere naturale $a$ şi $b$. Metoda de determinare a cmmdc poate fi descrisă în *limbaj natural* după cum urmează.

Se împarte $a$ la $b$ şi se reţine restul $r$. Se consideră ca nou deîmpărţit vechiul împărţitor şi ca nou împarţitor restul obţinut la împărţirea anterioară. Operaţia de împărţire continuă până se obţine un rest nul. Ultimul rest nenul (care a fost şi ultimul împărţitor) reprezintă rezultatul.

Se observă că metoda descrisă îndeplineşte proprietăţile unui algoritm: poate fi aplicată oricărei perechi de numere naturale iar numărul de prelucrări este finit (după un număr finit de împărţiri se ajunge la un rest nul).

De asemenea se observă că prelucrarea principală a algoritmului este una repetitivă, condiţia utilizată pentru a analiza dacă s-a terminat prelucrarea fiind egalitatea cu zero a restului.

2. *Schema lui Horner*. Permite determinarea câtului şi restului împărţirii unui polinom $P[X] = a_nX^n + a_{n-1}X^{n-1} + ... + a_1X + a_0 = 0$ la un binom de forma $X - b$.

O modalitate simplă de a descrie metoda de rezolvare este schema următoare:

| | $a_n$ | $a_{n-1}$ | ... | $a_k$ | ... | $a_2$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|---|---|---|
| $b$ | $a_n$ | $bc_{n-1} + a_{n-1}$ | ... | $bc_k + a_k$ | ... | $bc_2 + a_2$ | $bc_1 + a_1$ | $bc_0 + a_0$ |
| | $\Downarrow$ | $\Downarrow$ | ... | $\Downarrow$ | ... | $\Downarrow$ | $\Downarrow$ | $\Downarrow$ |
| | $c_{n-1}$ | $c_{n-2}$ | ... | $c_{k-1}$ | ... | $c_1$ | $c_0$ | $P[b]$ |

Valorile $c_{n-1}, c_{n-2}, ..., c_1, c_0$ reprezintă coeficienţii câtului, iar ultima valoare calculată reprezintă valoarea restului (valoarea polinomului calculată în $b$).

Şi în acest caz prelucrarea principală este una repetitivă constând în evaluarea expresiei $bc_k + a_k$ pentru $k$ luând, în această ordine, valorile $n-1, n-2, ..., 2, 1, 0$.

### 3.3.2 Scheme logice

Scrierea unui program pornind de la un algoritm descris într-un limbaj mai mult sau mai puţin riguros, ca în exemplele de mai sus, este dificilă întrucât nu sunt puşi în evidenţă foarte clar paşii algoritmului.

Modalităţi intermediare de descriere a algoritmilor, între limbajul natural sau cel matematic şi un limbaj de programare, sunt *schemele logice* şi *limbajele algoritmice*.

> *Schemele logice* sunt descrieri grafice ale algoritmilor în care fiecărui pas i se ataşează un simbol grafic, numit *bloc*, iar modul de înlănţuire a blocurilor este specificat prin segmente orientate.

Schemele logice au avantajul că sunt sugestive dar şi dezavantajul că pot deveni dificil de urmărit în cazul unor prelucrări prea complexe. Acest dezavantaj, dar şi evoluţia modului de concepere a programelor, fac ca schemele logice să fie din ce în ce mai puţin folosite (în favoarea limbajelor algoritmice).

### 3.3.3 Pseudocod

Un *limbaj algoritmic* este o notaţie care permite exprimarea logicii algoritmilor într-un mod formalizat fără a fi necesare reguli de sintaxă riguroase, ca în cazul limbajelor de programare.

Un limbaj algoritmic mai este denumit şi *pseudocod*. Un algoritm descris în pseudocod conţine atât enunţuri care descriu operaţii ce pot fi traduse direct într-un limbaj de programare (unui enunţ în limbaj algoritmic îi corespunde o instrucţiune în program) cât şi enunţuri ce descriu prelucrări ce urmează a fi detaliate abia în momentul scrierii programului.

Nu există un anumit standard în elaborarea limbajelor algoritmice, fiecare programator putând să conceapă propriul pseudocod, cu condiţia ca acesta să permită o descriere clară şi neambiguă a algoritmilor. Se poate folosi sintaxa limbajului de programare preferat, în care apar enunţuri de prelucrări. De exemplu:

**for** fiecare vârf $v$ din $V$
**{**
  culoare[v] = alb;
  distanta[v] = infinit;
  predecesor[v]=-1;
**}**

## 3.4 Limbaj algoritmic

În continuare prezentăm un exemplu de limbaj algoritmic.

### 3.4.1 Declararea datelor

*Datele simple* se declară sub forma:

<tip> <nume>;

unde <tip> poate lua una dintre valorile: **byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**.

Tablourile unidimensionale se declară sub forma:

<tip> <nume> $[n_1..n_2]$;

Elementele vectorului pot fi accesate cu ajutorul unui indice, care poate lua valori între $n_1$ şi $n_2$, sub forma:

<nume>[i]

unde $i$ poate lua orice valoare între $n_1$ şi $n_2$.

În cazul tablourilor bidimensionale, o declaraţie de forma:

<tip> <nume>$[m_1..m_2][n_1..n_2]$;

specifică o matrice cu $m_2 - m_1 + 1$ linii şi $n_2 - n_1 + 1$ coloane. Fiecare element se specifică prin doi indici:

<nume>[i][j]

unde $i$ reprezintă indicele liniei şi poate avea orice valoare între $m_1$ şi $m_2$ iar $j$ reprezintă indicele coloanei şi poate avea orice valoare între $n_1$ şi $n_2$.

### 3.4.2 Operaţii de intrare/ieşire

Preluarea valorilor pentru datele de intrare este descrisă sub forma:

**read** $v_1$, $v_2$, ...;

unde $v_1$, $v_2$, ... sunt nume de variabile.

Afişarea rezultatelor este descrisă sub forma:

**write** $e_1$, $e_2$, ...;

unde $e_1$, $e_2$, ... sunt expresii (în particular pot fi constante sau variabile).

**Operaţia de atribuire**. Operaţia de atribuire a unei valori către o variabilă se descrie prin:

v = <expresie>;

unde $v$ este un nume de variabilă, <expresie> desemnează o expresie aritmetică sau logică, iar "=" este *operatorul de atribuire*. Pentru acesta din urmă pot fi folosite şi alte simboluri, ca de exemplu ":=" sau "←". Expresiile pot fi descrise conform regulilor utilizate în matematică.

### 3.4.3 Prelucrări liniare

O secvenţă de prelucrări se descrie în modul următor:

<prel_1>;
<prel_2>;
...
<prel_n>;

sau

<prel_1>; <prel_2>; ... <prel_n>;

O astfel de scriere indică faptul că în momentul execuţiei prelucrările se efectuează în ordinea în care sunt specificate.

### 3.4.4 Prelucrări alternative

O prelucrare *alternativă completă* (cu două ramuri) este descrisă prin:

**if** <condiţie> <prel_1> **else** <prel_2>;

sau sub forma

**if** <condiţie> **then** <prel_1> **else** <prel_2>;

unde <condiţie> este o *expresie relaţională*. Această prelucrare trebuie înţeleasă în modul următor: dacă condiţia este *adevărată* atunci se efectuează prelucrarea <prel_1>, *altfel* se efectuează <prel_2>.

O prelucrare *alternativă cu o singură ramură* se descrie prin:

**if** <condiţie> <prel>;

sau

**if** <condiţie> **then** <prel>;

iar execuţia ei are următorul efect: *dacă* condiţia este satisfacută atunci se efectuează prelucrarea specificată, altfel nu se efectuează nici o prelucrare ci se trece la următoarea prelucrare a algoritmului.

### 3.4.5 Prelucrări repetitive

Prelucrările repetitive pot fi de trei tipuri:

- cu *test iniţial*,
- cu *test final* şi
- cu *contor*.

Prelucrarea *repetitivă cu test iniţial* se descrie prin: Prelucrarea *repetitivă cu test iniţial* se descrie prin:

**while** <condiţie> <prel>;

sau

**while** <condiţie> **do** <prel>;

În momentul execuţiei, *atât timp cât condiţia este adevarată*, se va executa instrucţiunea. Dacă condiţia nu este la început satisfăcută, atunci instrucţiunea nu se efectuează niciodată.

Prelucrarea *repetitivă cu test final* se descrie prin:

**do** <prel> **while** <condiţie>;

Prelucrarea se repetă până când condiţia specificată devine falsă. În acest caz prelucrarea se efectuează cel puţin o dată, chiar dacă condiţia nu este satisfacută la început.

Prelucrarea *repetitivă cu contor* se caracterizează prin repetarea prelucrării de un număr prestabilit de ori şi este descrisă prin:

**for** $i = i_1, i_2, ..., i_n$ <prel>;

sau

**for** $i = i_1, i_2, ..., i_n$ **do** <prel>;

unde $i$ este variabila contor care ia, pe rând, valorile $i_1, i_2, ..., i_n$ în această ordine, prelucrarea fiind efectuată pentru fiecare valoare a contorului.

Alte forme utilizate sunt:

**for** $i = v_i$ **to** $v_f$ **do** <prel>;

în care contorul ia valori consecutive crescătoare între $v_i$ şi $v_f$, şi

**for** $i = v_i$ **downto** $v_f$ **do** <prel>;

în care contorul ia valori consecutive descrescătoare între $v_i$ şi $v_f$.

### 3.4.6 Subalgoritm

În cadrul unui algoritm poate să apară necesitatea de a specifica de mai multe ori şi în diferite locuri un grup de prelucrări. Pentru a nu le descrie în mod repetat ele pot constitui o unitate distinctă, identificabilă printr-un nume, care este numită *subalgoritm*. Ori de câte ori este necesară efectuarea grupului de prelucrări din cadrul subalgoritmului se specifică numele acestuia şi, eventual, datele curente asupra cărora se vor efectua prelucrarile. Această acţiune se numeşte *apel al subalgoritmului*, iar datele specificate alături de numele acestuia şi asupra cărora se efectuează prelucrarile se numesc *parametri*.

În urma traducerii într-un limbaj de programare un subalgoritm devine un subprogram.

Un subalgoritm poate fi descris în felul următor:

```
<nume_subalg> (<tip> <nume_p1>, <tip> <nume_p2>, ... )
{
    ...
    /* prelucrări specifice subalgoritmului */
    ...
    return <nume_rezultat>;
}
```

unde <nume_subalg> reprezintă numele subalgoritmului iar nume_p1, nume_p2, ... reprezintă numele parametrilor. Ultimul enunţ, prin care se returnează rezultatul calculat în cadrul subalgoritmului, este optional.

*Modul de apel* depinde de modul în care subalgoritmul returnează rezultatele sale. Dacă subalgoritmul returnează efectiv un rezultat, printr-un enunţ de forma

**return** <nume_rezultat>;

atunci subalgoritmul se va apela în felul următor:

v=<nume_subalg>(nume_p1, nume_p2, ...);

Aceşti subalgoritmi corespund subprogramelor de tip *funcţie*.

Dacă în subalgoritm nu apare un astfel de enunţ, atunci el se va apela prin:

<nume_subalg>(nume_p1, nume_p2, ...);

variantă care corespunde subprogramelor de tip *procedură*.

*Observaţie*. Prelucrările care nu sunt detaliate în cadrul algoritmului sunt descrise în *limbaj natural* sau *limbaj matematic*. Comentariile suplimentare vor fi cuprinse între /* şi */. Dacă pe o linie a descrierii algoritmului apare simbolul `//` atunci tot ce urmează după acest simbol, pe aceeaşi linie cu el, este interpretat ca fiind un comentariu (deci, nu reprezintă o prelucrare a algoritmului).

### 3.4.7 Probleme rezolvate

1. Algoritmului lui Euclid.
Descrierea în pseudocod a algoritmului lui Euclid este următoarea:

```
int a, b, d, i, r;
read a, b;
if (a<b) { d=a; i=b; } else { d=b; i=a; };
r = d % i;
while (r ! = 0) { d=i; i=r; r=d % i; };
write i;
```

2. Schema lui Horner.
Descrierea în pseudocod a schemei lui Horner este următoarea:

```
int n, a, b, i;
read n, a, b;
int a[0..n], c[0..n-1];
for i=n,0,-1 read a[i];
c[n-1]=b*a[n];
for i=1,n-1 c[n-i-1]=b*c[n-i]+a[n-i];
val:=b*c[1]+a[1];
write val;
```

3. Conversia unui număr natural din baza 10 în baza 2.
Fie $n$ un număr întreg pozitiv. Pentru a determina cifrele reprezentarii în baza doi a acestui număr se poate folosi următoarea metodă:
Se împarte $n$ la 2, iar restul va reprezenta cifra de rang 0. Câtul obţinut la împartirea anterioară se împarte din nou la 2, iar restul obţinut va reprezenta cifra de ordin 1 ş.a.m.d. Secvenţa de împărţiri continuă pînă la obţinerea unui cât nul.
Descrierea în pseudocod a acestui algoritm este:

```
int n, d, c, r;
read n;
d = n;
c = d / 2;      /* câtul împărţirii întregi a lui d la 2 */
r = d % 2;      /* restul împărţirii întregi a lui d la 2 */
write r;
while (c != 0) {
    d = c;
    c = d / 2;      /* câtul împărţirii întregi a lui d la 2 */
    r = d % 2;      /* restul împărţirii întregi a lui d la 2 */
    write r;
}
```

4. Conversia unui număr întreg din baza 2 în baza 10.

Dacă $b_k b_{k-1}...b_1 b_0$ reprezintă cifrele numărului în baza 2, atunci valoarea în baza 10 se obţine efectuînd calculul:

$$(b_k b_{k-1}...b_1 b_0)_{10} = b_k 2^k + b_{k-1} 2^{k-1} + ... + b_1 2 + b_0$$

Deşi calculul de mai sus este similar cu evaluarea pentru $X = 2$ a polinomului

$$P[X] = b_k X^k + b_{k-1} X^{k-1} + ... + b_1 X + b_0$$

prelucrare pentru care ar putea fi folosit algoritmul corespunzător schemei lui Horner, în continuare prezentăm o altă variantă de rezolvare a acestei probleme, care foloseşte un subalgoritm pentru calculul puterilor unui număr întreg:

```
int k, i, s;
read k;
int b[0..k];
read b;
s = 0;
for i=0,k s = s+b[i] * putere(2,i);
write s;
```

```
putere(int a, int n)
{
    int i, p;
    p = 1;
    for i=2,n p = p*a;
    return p;
}
```

5. Să se scrie un algoritm pentru determinarea tuturor divizorilor naturali ai unui număr întreg.

*Rezolvare.* Fie $n$ numărul ai cărui divizori trebuie determinaţi. Evident 1 şi $|n|$ sunt divizori ai lui $n$. Pentru a determina restul divizorilor este suficient ca aceştia să fie căutaţi printre elementele mulţimii $\{2, 3, ..., [|n|]\}$ cu $[x]$ desemnând partea întreagă a lui $x$.

Algoritmul poate descris în modul următor:

```
int n, d;
read n;
write 1; /* afişarea primului divizor */
for d = 2, [|n|/2]
    if (d divide pe n) then write d;
write |n| /* afişarea ultimului divizor */
```

6. Să se scrie un algoritm pentru determinarea celui mai mare element dintr-un şir de numere reale.

*Rezolvare.* Fie $x_1, x_2, ..., x_n$ şirul analizat. Determinarea celui mai mare element constă în iniţializarea unei variabile de lucru $max$ (care va conţine valoarea maximului) cu $x_1$ şi compararea acesteia cu fiecare dintre celelalte elemente ale şirului. Dacă valoarea curentă a şirului, $x_k$, este mai mare decât valoarea variaabilei $max$ atunci acesteia din urmă i se va da valoarea $x_k$. Astfel, după a $k-1$ comparaţie variabila $max$ va conţine valoarea maximă din subşirul $x_1, x_2, ..., x_k$.

Algoritmul poate fi descris în modul următor:

```
int k, n;
read n;
double x[1..n], max;      /* vectorul şi variabila de lucru */
read x;                   /* preluarea elementelor şirului */
max = x[1];
for k = 2, n
    if (max < x[k]) then max = x[k];
write max;
```

7. Să se aproximeze, cu precizia $\varepsilon$, limita şirului

$$s_n = \sum_{k=0}^{n} \frac{1}{k!}.$$

*Rezolvare.* Calculul aproximativ (cu precizia $\varepsilon$) al limitei şirului $s_n$ constă în calculul sumei finite $s_k$, unde ultimul termen al sumei, $t_k = \frac{1}{k!}$, are proprietatea $t_k < \varepsilon$. Întrucât $t_{k+1} = \frac{t_k}{k+1}$, această relaţie va fi folosită pentru calculul valorii termenului curent (permiţând micşorarea numărului de calcule).

```
double eps, t, s;
int k;
k=1; /* iniţializare indice */
t=1; /* iniţializare termen */
s=1; /* iniţializare suma */
do {
    s=s+t; /* adăugarea termenului curent */
    k=k+1;
    t=t/k; /* calculul următorului termen */
} while (t ≥ eps);
s=s+t; (* adăugarea ultimului termen *)
write s;
```

8. Fie $A$ o matrice cu $m$ linii şi $n$ coloane, iar $B$ o matrice cu $n$ linii şi $p$ coloane, ambele având elemente reale. Să se determine matricea produs $C = A \times B$.

*Rezolvare.* Matricea $C$ va avea $m$ linii şi $p$ coloane, iar fiecare element se determină efectuând suma:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}, \qquad 1 \le i \le m,\ 1 \le j \le p.$$

În felul acesta calculul elementelor matricei $C$ se efectuează prin trei cicluri imbricate (unul pentru parcurgerea liniilor matricei $C$, unul pentru parcurgerea coloanelor matricei $C$, iar unul pentru efectuarea sumei specificate mai sus).

```
int m, n, p;      /* dimensiunile matricelor */
read m, n, p;
double a[1..m][1..n], b[1..n][1..p], c[1..m][1..p];      /* matrice */
int i, j, k; /* indici */
read a;      /* citirea matricei a */
read b;      /* citirea matricei b */
for i=1,m
    for j=1,p {
        c[i,j]=0;
        for k=1,n c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
write c;
```

### 3.4.8 Probleme propuse

1. Fie $D$ o dreaptă de ecuaţie $ax+by+c=0$ şi (C) un cerc de centru $O(x_0, y_0)$ şi rază $r$. Să se stabilească poziţia dreptei faţă de cerc.

*Indicaţie.* Se calculează distanţa de la centrul cercului la dreapta $D$ utilizând formula:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

Dacă $d \geq r+\varepsilon$ atunci dreapta este exterioară cercului, dacă $d \leq r-\varepsilon$ atunci dreapta este secantă, iar dacă $r - \varepsilon < d < r + \varepsilon$ atunci este tangentă (la implementarea egalitatea între două numere reale ...).

2. Să se genereze primele $n$ elemente ale şirurilor $a_k$ şi $b_k$ date prin relaţiile de recurenţă:

$$a_{k+1} = \frac{5a_k + 3}{a_k + 3}, \quad b_k = \frac{a_k + 3}{a_k + 1}, \quad k \geq 0, a_0 = 1.$$

3. Să se determine rădăcina pătrată a unui numŭr real pozitiv $a$ cu precizia $\varepsilon = 0.001$, folosind relaţia de recurenţă:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right), \quad x_1 = a.$$

Precizia se consideră atinsă când $|x_{n+1} - x_n| < \varepsilon$.

4. Fie $A$ o matrice pătratică de dimensiune $n$. Să se transforme matricea $A$, prin interrschimbări de linii şi de coloane, astfel încât elementele de pe diagonala principală să fie ordonate crescător.

5. Să se determine cel mai mare divizor comun al unui şir de numere întregi.

6. Să se calculeze coeficienţii polinomului

$$P[X] = (aX + b)^n, \quad a, b \in \mathbb{Z}, n \in \mathbb{N}.$$

7. Fie $A$ o matrice pătratică. Să se calculeze suma elementelor din fiecare zonă (diagonala principală, diagonala secundară, etc.) marcată în figura următoare:

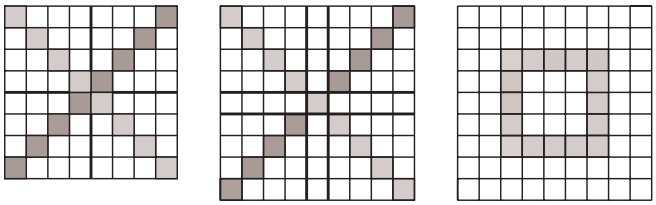



Figura 3.1: Zone în matrice pătratică

8. Fie $x_1, x_2, ..., x_n \in \mathbb{Z}$ rădăcinile unui polinom cu coeficienţi întregi:

$$P[X] = a_n X^n + a_{n-1} X^{n-1} + ... + a_1 X + a_0.$$

Să se determine coeficienţii polinomului.

9. Să se determine toate rădăcinile raţionale ale polinomului $P[X]$ care are coeficienţi întregi.

140. Fie $[P_1, P_2, ..., P_n]$ un poligon convex dat prin coordonatele carteziene ale vârfurilor sale (în ordine trigonometrică). Să se calculeze aria poligonului.

11. Fie $f : [a, b] \to \mathbb{R}$ o funcţie continuă cu proprietatea că există un unic $\xi \in (a, b)$ care are proprietatea că $f(\xi) = 0$. Să se aproximeze $\xi$ cu precizia $\varepsilon = 0.001$ utilizând metoda bisecţiei.

12. Fie $P$ şi $Q$ polinoame cu coeficienţi întregi. Să se determine toate rădăcinile raţionale comune celor două polinoame.

13. Să se determine toate numerele prime cu maxim 6 cifre care rămân prime şi după "răsturnarea" lor (răsturnatul numărului $\overline{abcdef}$ este $\overline{fedcba}$).

## 3.5 Instrucţiuni corespondente limbajului algoritmic

### 3.5.1 Declararea datelor

*Datele simple* se declară sub forma:

<tip> <nume>;

sau

<tip> <nume>= *literal*;

unde <tip> poate lua una dintre următoarele valori: **byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**. În exemplul următor sunt prezentate câteva modalităţi de declarare pentru diferite tipuri de variabile.

```
class Literali
{
   public static void main(String args[])
   {
      long   l1   =  5L;
      long   l2  =  12l;
      int    i1hexa  =  0x1;
      int    i2hexa  =  0X1aF;
      int    i3octal  =  01;
      long   i4octal  =  012L;
      long   i5LongHexa  =  0xAL;
      float  f1 = 5.40F;
      float  f2 = 5.40f;
      float  f3 = 5.40e2f;
      float  f4 = 5.40e+12f;
      float  f5 = 5.40;       // da eroare, trebuie cast
      double d1 = 5.40;       // implicit este double !
      double d2 = 5.40d;
      double d3 = 5.40D;
      double d4 = 5.40e2;
      double d5 = 5.40e+12d;
      char c1 = 'r';
      char c2 = '\u4567';
   }
}
```

Java defineşte mai multe tipuri primitive de date. Fiecare tip are o anumită dimensiune, care este independentă de caracteristicile maşinii gazdă. Astfel, spre deosebire de C/C++, unde un întreg poate fi reprezentat pe 16, 32 sau 64 de biţi, în funcţie de arhitectura maşinii, o valoare de tip întreg în Java va ocupa întotdeauna 32 de biţi, indiferent de maşina pe care rulează. Această consecvenţă este esenţială deoarece o aceeaşi aplicaţie va trebui să ruleze pe maşini cu arhitectură pe 16, 32 sau 64 de biţi şi să producă acelaşi rezultat pe fiecare maşină în parte.

| Tip | Dimensiune (octeţi) | Valoare minima | Valoare maxima | Valoare initiala | Cifre semnificative |
|---|---|---|---|---|---|
| byte | 1 | $-2^7$ | $2^7-1$ | 0 | |
| short | 2 | $-2^{15}$ | $2^{15}-1$ | 0 | |
| int | 4 | $-2^{31}$ | $2^{31}-1$ | 0 | |
| long | 8 | $-2^{63}$ | $2^{63}-1$ | 0 | |
| float | 4 | +-1.4E-45 | +-3.4E+38 | 0 | 6-7 |
| double | 8 | +-4.94E-324 | +-1.79E+308 | 0 | 14-15 |
| boolean | 1 | | | $false$ | |
| char | 2 | | | $null$ | |

Tabelul 3.1: Tipurile primitive de date în Java

*Variabilele* pot fi *iniţializate* la *declararea* lor sau în momentul utilizării lor efective. Dacă valoarea nu este specificată explicit atunci variabila se iniţializează cu o valoare iniţială implicită. Tabelul anterior prezintă câteva exemple în acest sens.

Conversiile între diferitele tipuri sunt permise (acolo unde au semnificaţie). Se vede din tabel că unele tipuri de variabile au posibilitatea să reprezinte un spectru mai mare de numere decât altele.

În afara tipurilor de bază, limbajul Java suportă şi tipuri de date create de utilizator, de pildă variabile de tip *clasă*, *interfaţă* sau *tablou*. Ca şi celelalte variabile, dacă nu sunt explicit iniţializate, valoarea atribuită implicit este *null*.

*Modificatorul* `static` este folosit pentru a specifica faptul că variabila are o singură valoare, comună tuturor instanţelor clasei în care ea este declarată. Modificarea valorii acestei variabile din interiorul unui obiect face ca modificarea să fie vizibilă din celelalte obiecte. *Variabilele statice* sunt iniţializate la *încărcarea codului* specific unei clase şi există chiar şi dacă nu există nici o instanţă a clasei respective. Din această cauză, ele pot fi folosite de *metodele statice*.

Tablourile unidimensionale se declară sub forma:

<tip>[ ] <nume> =new <tip>[$n$];

sau

<tip> <nume>[ ] =new <tip>[$n$];

Elementele vectorului pot fi accesate cu ajutorul unui indice, sub forma:

<nume>[i]

unde $i$ poate lua orice valoare între 0 şi $n-1$.

În cazul tablourilor bidimensionale, o declaraţie de forma:

<tip>[ ] [ ] <nume> = new <tip>[$m$][$n$];

sau

<tip> <nume> [ ] [ ] = new <tip>[$m$][$n$];

specifică o matrice cu $m$ linii şi $n$ coloane. Fiecare element se specifică prin doi indici:

<nume>[i][j]

unde $i$ reprezintă indicele liniei şi poate avea orice valoare între 0 şi $m-1$ iar $j$ reprezintă indicele coloanei şi poate avea orice valoare între 0 şi $n-1$.

### 3.5.2 Operaţii de intrare/ieşire

Preluarea unei valori de tip *int* de la tastatură se poate face sub forma:

```
BufferedReader br = new BufferedReader(
                    new InputStreamReader(System.in));
int vi=Integer.parseInt(br.readLine());
```

iar dintr-un fişier (de exemplu `fis.in`), sub forma:

```
StreamTokenizer st = new StreamTokenizer(
                     new BufferedReader(
                     new FileReader("fis.in")));
st.nextToken(); int vi = (int) st.nval;
```

Scrierea valorii unei variabile $v$ pe ecran se poate face sub forma:

```
System.out.print(v);
```

iar într-un fişier (de exemplu `fis.out`), sub forma:

```
PrintWriter out = new PrintWriter(
                  new BufferedWriter(
                  new FileWriter("fis.out")));
out.print(v);
out.close();
```

### 3.5.3 Prelucrări liniare

O secvenţă de prelucrări se descrie în modul următor:

<instr_1>;
<instr_2>;
...
<instr_n>;

sau

<instr_1>; <instr_2>; ... <instr_n>;

O astfel de scriere indică faptul că în momentul execuţiei instrucţiunile se efectuează în ordinea în care sunt specificate.

### 3.5.4 Prelucrări alternative

O prelucrare *alternativă completă* (cu două ramuri) este descrisă prin:

**if** (<condiţie>) <instr_1> **else** <instr_2>;

unde <condiţie> este o *expresie relaţională*. Această prelucrare trebuie înţeleasă în modul următor: dacă condiţia este *adevărată* atunci se efectuează prelucrarea <instr_1>, *altfel* se efectuează <instr_2>.

O prelucrare *alternativă cu o singură ramură* se descrie prin:

**if** (<condiţie>) <instr>;

iar execuţia ei are următorul efect: *dacă* condiţia este satisfacută atunci se efectuează instrucţiunea specificată, altfel nu se efectuează nici o prelucrare ci se trece la următoarea prelucrare a algoritmului.

### 3.5.5 Prelucrări repetitive

Prelucrările repetitive pot fi de trei tipuri:

- cu *test iniţial*,
- cu *test final* şi
- cu *contor*.

Prelucrarea *repetitivă cu test iniţial* se descrie prin:

**while** (<condiţie>) <instr>;

În momentul execuţiei, *atât timp cât condiţia este adevarată*, se va executa prelucrarea. Dacă condiţia nu este la început satisfăcută, atunci prelucrarea nu se efectuează niciodată.

Prelucrarea *repetitivă cu test final* se descrie prin:

**do** <instr> **while** (<condiţie>);

Instrucţiunea se repetă până când condiţia specificată devine falsă. În acest caz prelucrarea se efectuează cel puţin o dată, chiar dacă condiţia nu este satisfacută la început.

Prelucrarea *repetitivă cu contor* se caracterizează prin repetarea prelucrării de un număr prestabilit de ori şi este descrisă prin:

**for**(<instr1> ; <conditie>; <instr2>) <instr3>;

În general `<instr1>` reprezintă etapa de iniţializare a contorului, `<instr2>` reprezintă etapa de incrementare a contorului, `<instr3>` reprezintă instrucţiunea care se execută în mod repetat cât timp condiţia `<conditie>` are valoarea `true`.

### 3.5.6 Subprograme

În cadrul unui program poate să apară necesitatea de a specifica de mai multe ori şi în diferite locuri un grup de prelucrări. Pentru a nu le descrie în mod repetat ele pot constitui o unitate distinctă, identificabilă printr-un nume, care este numită *subprogram* sau, mai precis, *funcţie* (dacă returnează un rezultat) sau *procedură* (dacă nu returnează nici un rezultat). În Java *funcţiile* şi *procedurile* se numesc *metode*. Ori de câte ori este necesară efectuarea grupului de prelucrări din cadrul programului se specifică numele acestuia şi, eventual, datele curente asupra cărora se vor efectua prelucrarile. Această acţiune se numeşte *apel al subprogramului*, iar datele specificate alături de numele acestuia şi asupra cărora se efectuează prelucrarile se numesc *parametri*.

Un *subprogram* poate fi descris în felul următor:

```
<tipr> <nume_sp> (<tipp1> <numep1>, <tipp2> <numep2>, ... )
{
    ...
    /* prelucrări specifice subprogramului */
    ...
    return <nume_rezultat>;
}
```

unde <tipr> reprezintă tipul rezultatului returnat (`void` dacă subprogramul nu returnează nici un rezultat), <nume_sp> reprezintă numele subprogramului, iar numep1, numep2, ... reprezintă numele parametrilor. Ultimul enunţ, prin care se returnează rezultatul calculat în cadrul subprogramului, trebuie pus numai dacă <tipr> nu este `void`.

*Modul de apel* depinde de modul în care subprogramul returnează rezultatele sale. Dacă subprogramul returnează efectiv un rezultat, printr-un enunţ de forma

**return** <nume_rezultat>;

atunci subprogramul se va apela în felul următor:

v=<nume_sp>(nume_p1, nume_p2, ...);

Aceste subprograme corespund subprogramelor de tip *funcţie*.

Dacă în subprogram nu apare un astfel de enunţ, atunci el se va apela prin:

<nume_sp>(nume_p1, nume_p2, ...);

variantă care corespunde subprogramelor de tip *procedură*.

*Observaţie*. Prelucrările care nu sunt detaliate în cadrul algoritmului sunt descrise în *limbaj natural* sau *limbaj matematic*. Comentariile suplimentare vor fi cuprinse între /* şi */. Dacă pe o linie a descrierii algoritmului apare simbolul `//` atunci tot ce urmează după acest simbol, pe aceeaşi linie cu el, este interpretat ca fiind un comentariu (deci, nu reprezintă o prelucrare a programului).

### 3.5.7 Probleme rezolvate

1. *Descompunere Fibonacci*. Să se descompună un număr natural, de cel mult 18-19 cifre, în sumă de cât mai puţini termeni Fibonacci.

*Rezolvare*: Programul următor calculează şi afişează primii 92 de termeni din şirul Fibonacci (mai mult nu este posibil fără *numere mari*!), şi descompune numărul `x` introdus de la tastatură. Metoda `static int maxFibo ( long nr )` returnează indicele celui mai mare element din şirul lui Fibonacci care este mai mic sau egal cu parametrul `nr`.

```
import java.io.*;
class DescFibo
{
  static int n=92;
  static long[] f=new long[n+1];

  public static void main (String[]args) throws IOException
  {
    long x,y;
    int iy, k, nrt=0;

    BufferedReader br = new BufferedReader(
                        new InputStreamReader(System.in));
    System.out.print("x = ");
    x=Long.parseLong(br.readLine());
    f[0]=0; f[1]=1; f[2]=1;
    for(k=3;k<=n;k++) f[k]=f[k-1]+f[k-2];
    for(k=0;k<=n;k++) System.out.println(k+" : "+f[k]);
    System.out.println("     "+Long.MAX_VALUE+" = Long.MAX_VALUE");
```

```
    System.out.println("x = "+x);
    while(x>0)
    {
      iy=maxFibo(x);
      y=f[iy];
      nrt++;
      System.out.println(nrt+" : "+x+" f["+iy+"] = "+y);
      x=x-y;
    }
  }

  static int maxFibo(long nr)
  {
    int k;
    for(k=1;k<=n;k++) if (f[k]>nr) break;
    return k-1;
  }
}
```

De exemplu, pentru `x = 5678` pe ecran apare:

```
1 : 5678 f[19] = 418
2 : 1497 f[16] = 987
3 : 510 f[14] = 377
4 : 133 f[11] = 89
5 : 44 f[9] = 34
6 : 10 f[6] = 8
7 : 2 f[3] = 2
```

2. Fie $S_n = x_1^n + x_2^n$ unde $x_1$ şi $x_2$ sunt rădăcinile ecuaţiei cu coeficienţi întregi $ax^2 + bx + c = 0$ (vom considera $a = 1$!). Să se afişeze primii 10 termeni ai şirului $S_n$ şi să se precizeze în dreptul fiecărui termen dacă este număr prim, iar dacă nu este număr prim să se afişeze descompunerea în factori.

*Rezolvare*:

```
class e02
{
  public static void main(String[] args)
  {
    int a, b, c, nnp=0, s, p, n=10, k;
    long[] ss=new long[n+1];

    a=1;b=1;c=2;
    s=-b/a;
```

```
    p=c/a;
    ss[1]=s;
    ss[2]=s*s-2*p;
    for(k=3;k<=n;k++) ss[k]=s*ss[k-1]-p*ss[k-2];
    for(k=1;k<=n;k++)
      if(esteprim(Math.abs(ss[k])))
        System.out.println(k+" : "+ss[k]+" PRIM "+(++nnp));
      else
      {
        System.out.print(k+" : "+ss[k]+" = ");
        descfact(Math.abs(ss[k]));
      }
    System.out.println("nnp = "+nnp);
  }// main

  static void descfact(long nr)
  {
    long d=2;
    if((nr==0)||(nr==1)){System.out.println(); return;}
    while(nr%d==0){System.out.print(d+""); nr=nr/d;}
    d=3;
    while((d*d<=nr)&&(nr!=1))
    {
      while(nr%d==0){System.out.print(d+" "); nr=nr/d;}
      d=d+2;
    }
    if(nr!=1) System.out.println(nr);
      else System.out.println();
  }

  static boolean esteprim(long nr)
  {
    if((nr==0)||(nr==1)) return false;
    if((nr==2)||(nr==3)) return true;
    if(nr%2==0) return false;
    long d=3;
    while((nr%d!=0)&&(d*d<=nr)) d=d+2;
    if(nr%d==0) return false; else return true;
  }
}// class
```

Pe ecran apar următoarele rezultate:

```
1 : -1 =
2 : -3 PRIM 1
```

```
3 : 5 PRIM 2
4 : 1 =
5 : -11 PRIM 3
6 : 9 = 3 3
7 : 13 PRIM 4
8 : -31 PRIM 5
9 : 5 PRIM 6
10 : 57 = 3 19
nnp = 6
Press any key to continue...
```

3. Se consideră funcţia $f(x) = P(x)e^{\alpha x}$ unde $P(x)$ este un polinom de grad $n$ cu coeficienţi întregi. Să se afişeze toate derivatele până la ordinul $m$ ale funcţiei $f$, şi, în dreptul coeficienţilor polinoamelor care apar în aceste derivate, să se precizeze dacă respectivul coeficient este număr prim, iar dacă nu este număr prim să se afişeze descompunerea în factori. De asemenea, să se afişeze care este cel mai mare număr prim care apare, şi care este ordinul derivatei în care apare acest cel mai mare număr prim.

*Rezolvare*: Derivata funcţiei $f$ are forma $Q(x)e^{\alpha x}$ unde $Q$ este un polinom de acelaşi grad cu polinomul $P$. Toată rezolvarea problemei se reduce la determinarea coeficienţilor polinomului $Q$ în funcţie de coeficienţii polinomului $P$.

```
class e03
{
  static long npmax=1,pmax=0;

  public static void main(String[] args)
  {
    int n=7, m=10, alfa=1, k;
    long[] p=new long[n+1];
    p[7]=1; p[3]=1; p[0]=1;
    afisv(p,0);
    for(k=1;k<=m;k++)
    {
      System.out.print("derivata = "+k);
      p=deriv(p,alfa);
      afisv(p,k);
    }
    System.out.println(npmax+"  "+pmax);
    System.out.println("GATA!!!");
  }

  static long[] deriv(long[] a,int alfa)
```

```
{
  int n=a.length-1, k;
  long[] b=new long[n+1];
  b[n]=a[n]*alfa;
  for(k=0;k<=n-1;k++) b[k]=(k+1)*a[k+1]+a[k]*alfa;
  return b;
}

static void afisv(long[] x,int ppp)
{
  int n=x.length-1;
  int i;
  System.out.println();
  for(i=n;i>=0;i--)
  if(esteprim(Math.abs(x[i])))
  {
    System.out.println(i+" : "+x[i]+" PRIM ");
    if(npmax<Math.abs(x[i]))
    {
      npmax=Math.abs(x[i]);
      pmax=ppp;
    }
  }
  else
  {
    System.out.print(i+" : "+x[i]+" = ");
    descfact(Math.abs(x[i]));
  }
  System.out.println();
}

static void descfact(long nr)
{
  long d=2;
  if((nr==0)||(nr==1))
  {
    System.out.println();
    return;
  }
  while(nr%d==0)
  {
    System.out.print(d+" ");
    nr=nr/d;
  }
```

```
    d=3;
    while((d*d<=nr)&&(nr!=1))
    {
      while(nr%d==0)
      {
        System.out.print(d+" ");
        nr=nr/d;
      }
      d=d+2;
    }
    if(nr!=1) System.out.println(nr);
      else System.out.println();
  }

  static boolean esteprim(long nr)
  {
    if((nr==0)||(nr==1)) return false;
    if((nr==2)||(nr==3)) return true;
    if(nr%2==0) return false;
    long d=3;
    while((nr%d!=0)&&(d*d<=nr)) d=d+2;
    if(nr%d==0) return false; else return true;
  }
}// class
```

4. Rădăcini raţionale. Să se determine toate rădăcinile raţionale ale unei ecuaţii cu coeficienţi întregi.

*Rezolvare*: Se caută rădăcini raţionale formate din fracţii în care numărătorul este divizor al termenului liber iar numitorul este divizor al termenului dominant. Programul care urmează generează coeficienţii ecuaţiei, plecând de la fracţii date (ca rădăcini), şi apoi determină rădăcinile raţionale

```
class RadaciniRationale // generez p_i/q_i
{
  static int k=0;

  public static void main(String[] args)
  {
    int[] p={1,1,2,3, 3, 1}, q={2,3,3,2,-2,-1};
    int[] a=genPol(p,q);
    int n=a.length-1,alfa,beta;
    int moda0=Math.abs(a[0]),modan=Math.abs(a[n]);
    for(alfa=1;alfa<=moda0;alfa++)
```

```
    {
      if(moda0%alfa!=0) continue;
      for(beta=1;beta<=modan;beta++)
      {
        if(modan%beta!=0) continue;
        if(cmmdc(alfa,beta)!=1) continue;
        if (f(a,alfa,beta)==0)
          System.out.println("x["+(++k)+"] = "+alfa+"/"+beta+"  ");
        if (f(a,-alfa,beta)==0)
          System.out.println("x["+(++k)+"] = -"+alfa+"/"+beta+"  ");
      }// for beta
    }// for alfa
  }// main

  static int[] genPol(int[] a, int[] b) // X-a_i/b_i==>b_i X - a_i
  {
    int n=a.length;
    int[]  p={-a[0],b[0]},//p=b[0] X -a[0]
    q={13,13}; // q initializat "aiurea" - pentru dimensiune !
    afisv(p);
    for(int k=1;k<n;k++)
    {
      q[0]=-a[k];
      q[1]=b[k];
      p=pxq(p,q);
      afisv(p);
    }
    return p;
  }// genPol()

  static int[] pxq(int[] p,int[] q)
  {
    int gradp=p.length-1, gradq=q.length-1;
    int gradpq=gradp+gradq;
    int[] pq=new int[gradpq+1];
    int i,j,k;
    for(k=0;k<=gradpq;k++) pq[k]=0;
    for(i=0;i<=gradp;i++)
      for(j=0;j<=gradq;j++) pq[i+j]+=p[i]*q[j];
    return pq;
  }

  static int f(int[]a,int alfa, int beta)
  {
```

```
    int n=a.length-1,k,s=0;
    for(k=0;k<=n;k++) s+=a[k]*putere(alfa,k)*putere(beta,n-k);
    return s;
  }

  static int putere(int a, int n)
  {
    int p=1;
    for(int k=1;k<=n;k++) p*=a;
    return p;
  }

  static int cmmdc(int a, int b)
  {
    int d,i,c,r;
    if (a>b) {d=a; i=b;} else {d=b; i=a;}
    r=123; // ca sa inceapa while !!!
    while (r > 0){c=d/i; r=d%i; d=i; i=r;}
    return d;
  }

  static void afisv(int[] a)
  {
    for(int i=a.length-1;i>=0;i--) System.out.print(a[i]+" ");
    System.out.println();
  }// afisv()
}// class
```

5. Să se afişeze frecvenţa cifrelor care apar în

$$f(n) = \sum_{k=0}^{n} \frac{1}{2^k} C_{n+k}^{n}$$

neţinând cont de faptul că $f(n)$ are o expresie mult mai simplă, şi anume $2^n$. Suma trebuie calculată simulând operaţiile de adunare, înmulţire şi împărţire la 2, cu numere mari.

*Rezolvare*: Funcţia se pune sub forma:

$$f(n) = \frac{1}{2^n} \sum_{k=0}^{n} 2^{n-k} C_{n+k}^{n}$$

Se calculează suma, şi apoi se fac $n$ împărţiri succesive la 2.

```
class e05
```

```
{
  public static void main (String[]args)
  {
    int n, k;
    int[] s;
    int[] p;
    for(n=10;n<=12;n++)
    {
      s=nrv(0);
      for(k=0;k<=n;k++)
      {
        p=inm(comb(n+k,n),putere(2,n-k));
        s=suma(s,p);
      }
      afisv(s);
      for(k=1;k<=n;k++) s=impartLa2(s);
      System.out.print(n+" : ");
      afisv(s);
      fcifre(s);
    }
    System.out.println("GATA");
  }//main()

  static int[] impartLa2(int[] a)
  {
    int na,nb,k,t=0;
    na=a.length-1;
    if(a[na]==1) nb=na-1; else nb=na;
    int[] b=new int[nb+1];
    if(na==nb)
    for(k=na;k>=0;k--) {a[k]+=10*t; b[k]=a[k]/2; t=a[k]%2;}
    else
    {
      t=a[na];
      for(k=na-1;k>=0;k--){a[k]+=10*t; b[k]=a[k]/2; t=a[k]%2;}
    }
    return b;
  }

  static void fcifre(int[] x)
  {
    int i;
    int[] f=new int[10];
    for(i=0;i<x.length;i++) f[x[i]]++;
```

```
    System.out.println();
    for(i=0;i<=9;i++) System.out.println(i+" : "+f[i]);
    System.out.println();
  }

  static int[] suma(int[] x, int[] y)
  {
    int i, j, t, ncx=x.length, ncy=y.length, ncz;
    if(ncx>ncy) ncz=ncx+1; else ncz=ncy+1;
    int[] xx=new int[ncz];
    int[] yy=new int[ncz];
    int[] z=new int[ncz];
    for(i=0;i<ncx;i++) xx[i]=x[i];
    for(j=0;j<ncy;j++) yy[j]=y[j];
    t=0;
    for(i=0;i<ncz;i++){z[i]=xx[i]+yy[i]+t; t=z[i]/10; z[i]=z[i]%10;}
    if(z[ncz-1]!= 0) return z;
    else
    {
      int[]zz=new int[ncz-1];
      for(i=0;i<=ncz-2;i++) zz[i]=z[i];
      return zz;
    }
  }

  static int[] inm(int[]x,int[]y)
  {
    int t, n=x.length, m=y.length, i, j;
    int[][]a=new int[m][n+m];
    int[]z=new int[m+n];
    for(j=0;j<m;j++)
    {
      t=0;
      for(i=0;i<n;i++)
      {
        a[j][i+j]=y[j]*x[i]+t;
        t=a[j][i+j]/10;
        a[j][i+j]=a[j][i+j]%10;
      }
      a[j][i+j]=t;
    }
    t=0;
    for(j=0;j<m+n;j++)
    {
```

```
    z[j]=0;
    for(i=0;i<m;i++) z[j]=z[j]+a[i][j];
    z[j]=z[j]+t;
    t=z[j]/10;
    z[j]=z[j]%10;
  }
  if(z[m+n-1]!= 0) return z;
  else
  {
    int[]zz=new int[m+n-1];
    for(i=0;i<=m+n-2;i++)
    zz[i]=z[i];
    return zz;
  }
}

static void afisv(int[]x)
{
  int i;
  for(i=x.length-1;i>=0;i--) System.out.print(x[i]);
  System.out.print(" *** "+x.length);
  System.out.println();
}

static int[] nrv(int nr)
{
  int nrrez=nr, nc=0;
  while(nr!=0) {nc++; nr=nr/10;}
  int[]x=new int [nc];
  nr=nrrez;
  nc=0;
  while(nr!=0){x[nc]=nr%10; nc++; nr=nr/10;}
  return x;
}

static int[] putere (int a, int n)
{
  int[] rez;
  int k;
  rez=nrv(1);
  for(k=1;k<=n;k++) rez=inm(rez,nrv(a));
  return rez;
}
```

```
  static int[] comb (int n, int k)
  {
    int[] rez;
    int i, j, d;
    int[]x=new int[k+1];
    int[]y=new int[k+1];
    for(i=1;i<=k;i++) x[i]=n-k+i;
    for(j=1;j<=k;j++) y[j]=j;
    for(j=2;j<=k;j++)
    {
      for(i=1;i<=k;i++)
      {
        d=cmmdc(y[j],x[i]);
        y[j]=y[j]/d;
        x[i]=x[i]/d;
        if(y[j]==1) break;
      }
    }
    rez=nrv(1);
    for(i=1;i<=k;i++) rez=inm(rez,nrv(x[i]));
    return rez;
  }

  static int cmmdc (int a,int b)
  {
    int d,i,c,r;
    if (a>b) {d=a;i=b;} else{d=b;i=a;}
    while (i!=0){c=d/i; r=d%i; d=i; i=r;}
    return d;
  }
}// class
```

6. Să se afişeze $S(n,1), S(n,2), ..., S(n,m)$ (inclusiv suma cifrelor şi numărul cifrelor pentru fiecare numă) ştiind că

$$S(n+1,m) = S(n,m-1) + mS(n,m)$$

şi

$$S(n,1) = S(n,n) = 1, \forall n \geq m.$$

Se vor implementa operaţiile cu numere mari.

*Rezolvare*: Matricea de calcul este subdiagonală. Se completează cu 1 prima coloană şi diagonala principală, iar apoi se determină celelalte elemente ale matricei folosind relaţia dată (aranjată puţin altfel!). Matricea de calcul va avea de fapt trei

dimensiuni (numerele devin foarte mari, aşa că elementul $S_{i,j}$ trebuie să conţină vectorul cifrelor valorii sale).

```
class e06
{
  public static void main(String[] args)
  {
    int n=50, m=40, i, j;
    int[][][] s=new int[n+1][m+1][1];
    for(i=1;i<=n;i++)
    {
      if(i<=m) s[i][i]=nr2v(1);
      s[i][1]=nr2v(1);
      for(j=2;j<=min(i,m);j++)
        s[i][j]=suma(s[i-1][j-1],inm(nr2v(j),s[i-1][j]));
      if(i<=m) s[i][i]=nr2v(1);
    }
    for(i=1;i<=m;i++)
    {
      System.out.print("\n"+i+" : "+s[n][i].length+" ");
      afissumac(s[n][i]);
      afisv(s[n][i]);
    }
  }

  static int[] suma(int[] x,int[] y){...}
  static int[] nr2v(int nr){...}
  static int[] inm(int[]x, int[]y){...}
  static void afisv(int[]x){...}

  static void afissumac(int[]x)
  {
    int i,s=0;
    for(i=x.length-1;i>=0;i--) s+=x[i];
    System.out.print(s+" ");
  }

  static int min(int a, int b) { return (a<b)?a:b; }
}// class
```

Pe ecran apar următoarele valori (numerele devin foarte mari!):

```
1 : 1 1 1
2 : 15 64 562949953421311
```

```
3 : 24 102 119649664052358811373730
4 : 29 138 52818655359845224561907882505
5 : 33 150 74009586436825301627118813958 7625
6 : 37 170 11218727630940119874547782377128 16687
7 : 39 172 35571605929275246479706503801313 7686280
8 : 41 163 35041731132610098771332691525663865902850
9 : 43 189 13850225097959561846019070897007305096801 95
10 : 44 205 26154716515862881292012777396577993781727011
11 : 45 177 26723575409002161865117527704693137105019 4780
12 : 46 205 16193309449362797791543817458164280364412864 10
13 : 46 232 62389012762757848114928617948267375638892882 30
14 : 47 205 16132809270066494376125322988035691981158490930
15 : 47 162 29226457001965139089793853213126510270024300000
16 : 47 216 38400825365495544823847807988536071815780050940
17 : 47 198 37645241791600906804871080818625037726247519045
18 : 47 225 28189332813493454141899976735501798322277536165
19 : 47 165 16443993651925074352512402220900950019217097000
20 : 46 237 75979216068609869004544693940992771469987553 00
21 : 46 198 28202550285635061496579529546378130481727233 80
22 : 45 189 85122188307735663424162227664625917075162638 0
23 : 45 198 21109249414994737119560869609964510716814640 0
24 : 44 192 43397743800247894833556570977432285162431400
25 : 43 168 74538021532732000833796262348376254659125 00
26 : 43 186 10766896015976728016507126542097725743282 12
27 : 42 189 13154662736580840581381485825646536945608 0
28 : 41 155 13660054661277961013613328658015172843800
29 : 40 165 12105466866549001690105888404309633877 20
30 : 38 185 91860943867630642501164254978867961752
31 : 37 155 59851233855516250850900077938313625 60
32 : 36 164 33550607916361474458148864887018752 0
33 : 35 153 16204251384884158932677856617905110
34 : 33 144 67483341642571152248238137954496 0
35 : 32 126 24235536318546124501501767693750
36 : 30 135 75013568829210188677056801079 5
37 : 29 141 19983209983507514547524896035
38 : 27 132 45714934748917557373734424 5
39 : 25 114 89517797434964123149470 00
40 : 24 93 149377949042637543000150
```

7. Să se afişeze $B_1, B_2, ..., B_n$ ştiind că

$$B_{n+1} = \sum_{k=0}^{n} C_n^k B_k, B_0 = 1.$$

Se vor implementa operaţiile cu numere mari.

*Rezolvare*: Vectorul de calcul va avea de fapt două dimensiuni (numerele devin foarte mari, aşa că elementul $B_i$ trebuie să conţină vectorul cifrelor valorii sale).

```
class e07
{
  public static void main(String[] args)
  {
    int n=71; // n=25 ultimul care incape pe long
    int k,i;
    int[][] b=new int[n+1][1];
    int[] prod={1};

    b[0]=nr2v(1);
    for(i=1;i<=n;i++)
    {
      b[i]=nr2v(0);;
      for(k=0;k<=i-1;k++)
      {
        prod=inm(comb(i-1,k),b[k]);
        b[i]=suma(b[i],prod);
      }
      System.out.print(i+" : ");
      afisv(b[i]);
    }
    System.out.println("     "+Long.MAX_VALUE);
    System.out.println("... Gata ...");
  }

  static int[] suma(int[] x,int[] y){...}
  static int[] nr2v(int nr){...}
  static int[] inm(int[]x, int[]y){...}
  static void afisv(int[]x){...}

  static int[] comb(int n,int k)
  {
    int i,j,d;
    int[] rez;
    int[] x=new int[k+1];
    int[] y=new int[k+1];
    for(i=1;i<=k;i++) x[i]=n-k+i;
    for(j=1;j<=k;j++) y[j]=j;
    for(j=2;j<=k;j++)
      for(i=1;i<=k;i++)
```

```
      {
        d=cmmdc(y[j],x[i]);
        y[j]=y[j]/d;
        x[i]=x[i]/d;
        if(y[j]==1) break;
      }
    rez=nr2v(1);
    for(i=1;i<=k;i++) rez=inm(rez,nr2v(x[i]));
    return rez;
  }

  static int cmmdc(int a,int b) {...}
}
```

### 3.5.8 Probleme propuse

1. Fie $S_n = x_1^n + x_2^n + x_3^n$ unde $x_1$, $x_2$ şi $x_3$ sunt rădăcinile ecuaţiei cu coeficienţi întregi $ax^3 + bx^2 + cx + d = 0$ (vom considera $a = 1$!). Să se afişeze primii 10 termeni ai şirului $S_n$ şi să se precizeze în dreptul fiecărui termen dacă este număr prim, iar dacă nu este număr prim să se afişeze descompunerea în factori.

2. Să se afişeze frecvenţa cifrelor care apar în

$$f(n) = \sum_{k=0}^{n-1} C_{n-1}^k n^{n-1-k}(k+1)!$$

neţinând cont de faptul că $f(n)$ are o expresie mult mai simplă, şi anume $n^n$. Suma trebuie calculată simulând operaţiile cu numere mari.

3. Să se afişeze frecvenţa cifrelor care apar în

$$f(n) = n^{n-1} + \sum_{k=1}^{n-1} C_n^k k^{k-1}(n-k)^{n-k}$$

neţinând cont de faptul că $f(n)$ are o expresie mult mai simplă, şi anume $n^n$. Suma trebuie calculată simulând operaţiile cu numere mari.

4. Să se calculeze

$$f(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)...\left(1 - \frac{1}{p_m}\right)$$

unde $n = p_1^{i_1} p_2^{i_2} ... p_m^{i_m}$ reprezintă descompunerea în factori primi a lui $n$.

5. Să se calculeze

$$\phi(n) = card\left\{k \in \mathbb{N} / 1 \leq k \leq n, cmmdc(k,n) = 1\right\}.$$

6. Să se calculeze

$$f(n) = \sum_{d|n} \phi(n)$$

unde $\phi$ este funcţia de la exerciţiul anterior, neţinând cont de faptul că $f(n)$ are o expresie mult mai simplă, şi anume $n$.

7. Să se calculeze

$$f(n) = n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - ... + \frac{(-1)^n}{n!}\right).$$

8. Să se calculeze

$$f(m,n,\lambda_1,\lambda_2,...,\lambda_n) = \sum_{k=1}^{m} (-1)^{m-k} C_m^k \left(C_k^1\right)^{\lambda_1} \left(C_{k+1}^2\right)^{\lambda_2} ... \left(C_{k+n-1}^n\right)^{\lambda_n}.$$

9. Să se calculeze

$$g(m,n,\lambda_1,\lambda_2,...,\lambda_n) = \left(C_m^1\right)^{\lambda_1} \left(C_{m+1}^2\right)^{\lambda_2} ... \left(C_{m+n-1}^n\right)^{\lambda_n}$$

implementând operaţiile cu numere mari.

10. Să se calculeze

$$f(n) = \frac{1}{2^n}\left((2n)! - C_n^1 2(2n-1)! + C_n^2 2^2 (2n-2)! - ... + (-1)^n 2^n n!\right).$$

11. Să se calculeze

$$C_n = \frac{1}{n+1} C_{2n}^n$$

implementând operaţiile cu numere mari.

12. Să se afişeze $P(100,50)$ (inclusiv suma cifrelor şi numărul cifrelor) ştiind că

$$P(n+k,k) = P(n,1) + P(n,2) + ... + P(n,k)$$

şi

$$P(n,1) = P(n,n) = 1, \forall n \geq k \geq 1.$$

Se vor implementa operaţiile cu numere mari.

13. Să se determine cel mai mic număr natural $r$, astfel încât $p^r = e$, unde $p$ este o permutare dată şi $e$ este permutarea identică.

14. Să se afişeze $C_{100}$ ştiind că

$$C_n = \sum_{k=1}^{n} C_{k-1} C_{n-k}, C_0 = 1.$$

Se vor implementa operaţiile cu numere mari.

15. Să se afişeze $E_{100}$ ştiind că

$$En = E_2 E_{n-1} + E_3 E_{n-2} + ... + E_{n-1} E_2, E_1 = E_2 = 1.$$

Se vor implementa operaţiile cu numere mari.

16. Să se calculeze

$$S(n,m) = \frac{1}{m!} \sum_{k=0}^{m-1} (-1)^k C_m^k (m-k)^n$$

17. Să se afişeze $C_{100}$ ştiind că

$$C_n = \sum_{k=1}^{n} C_n^k F_k.$$

unde $F_k$ este termen Fibonacci. Se vor implementa operaţiile cu numere mari.

18. Să se afişeze $C_{100}$ ştiind că

$$C_n = \sum_{k=1}^{n} C_n^k 2^k F_k.$$

unde $F_k$ este termen Fibonacci. Se vor implementa operaţiile cu numere mari.

19. Să se determine puterea a zecea a unui polinom dat.

# Capitolul 4

# Analiza complexităţii algoritmilor

## 4.1 Scopul analizei complexităţii

În general există mai multi algoritmi care rezolvă aceeaşi problemă. Dorim să exprimăm *eficienţa algoritmilor* sub forma unui criteriu care să ne permită să alegem din mai mulţi algoritmi pe cel optim. Există mai multe moduri în care putem exprima *eficienţa*: prin timpul necesar pentru execuţia algoritmului sau prin alte resurse necesare (de exemplu memoria). În ambele cazuri însă, avem o dependenţă de dimensiunea cazului studiat.

Se pune problema de alegere a unei unităţi de măsură pentru a exprima eficienţa teoretică a unui algoritm. O importanţă deosebită în rezolvarea acestei probleme o are *principiul invarianţei*. Acesta ne arată că nu este necesar să folosim o astfel de unitate.

**Principiul invarianţei**: *două implementări diferite ale aceluiaşi algoritm nu diferă în eficienţă cu mai mult de o constantă multiplicativă.*

Implementarea unui algoritm presupune elementele legate de calculatorul folosit, de limbajul de programare si îndemânarea programatorului (cu condiţia ca acesta să nu modifice algoritmul). Datorită *principiului invarianţei* vom exprima eficienţa unui algoritm în limitele unei constante multiplicative.

Un algoritm este compus din mai multe *instrucţiuni*, care la rândul lor sunt compuse din mai multe *operaţii elementare*. Datorită *principiului invarianţei* nu ne interesează *timpul* de execuţie a unei *operaţii elementare*, ci numai *numărul lor*, dar ne interesează care şi ce sunt *operaţiile elementare*.

**Definiţia 1** *O operaţie elementară este o operaţie al cărui timp de execuţie poate fi mărginit superior de o constantă care depinde numai de particularitatea implementării (calculator, limbaj de programare etc).*

Deoarece ne interesează timpul de executie în limita unei constante multiplicative, vom considera doar numărul operaţiilor elementare executate într-un algoritm, nu şi timpul exact de execuţie al operaţiilor respective.

Este foarte important ce anume definim ca *operaţie elementară*. Este adunarea o operaţie elementara? Teoretic nu este, pentru că depinde de lungimea celor doi operanzi. Practic, pentru operanzi de lungime rezonabilă putem să considerăm că adunarea este o *operaţie elementară*. Vom considera în continuare că adunările, scăderile, înmulţirile, împărţirile, operaţiile modulo (restul împăarţirii întregi), operaţiile booleene, comparaţiile şi atribuirile sunt *operaţii elementare*.

Uneori eficienţa diferă dacă ţinem cont numai de unele operaţii elementare şi le ignorăm pe celelalte (de exemplu la sortare: comparaţia şi interschimbarea). De aceea în analiza unor algoritmi vom considera o anumită operaţie elementară, care este caracteristică algoritmului, ca operaţie barometru, neglijându-le pe celelalte.

De multe ori, timpul de executie al unui algoritm poate varia pentru cazuri de mărime identică. De exemplu la sortare, dacă introducem un şir de $n$ numere gata sortat, timpul necesar va cel mai mic dintre timpii necesari pentru sortarea oricarui alt şir format din $n$ numere. Spunem că avem de-a face cu *cazul cel mai favorabil*. Dacă şirul este introdus în ordine inversă, avem *cazul cel mai defavorabil* şi timpul va fi cel mai mare dintre timpii de sortare a şirului de $n$ numere.

Există algoritmi în care timpul de execuţie nu depinde de cazul considerat.

Dacă dimensiunea problemei este mare, îmbunătăţirea ordinului algoritmului este esenţială, în timp ce pentru timpi mici este sufcientă performanţa hardware.

Elaborarea unor algoritmi eficienţi presupune cunoştinţe din diverse domenii (informatică, matematică şi cunoştiinţe din domeniul căruia îi aparţine problema practică a cărui model este studiat, atunci când este cazul).

**Exemplul 1** *Elaboraţi un algoritm care returnează cel mai mare divizor comun (cmmdc) a doi termeni de rang oarecare din şirul lui Fibonacci.*

Şirul lui Fibonacci, $f_n = f_{n-1} + f_{n-2}$, este un exemplu de recursivitate în cascadă şi calcularea efectivă a celor doi termeni $f_m$ $f_n$, urmată de calculul celui mai mare divizor al lor, este total neindicată. Un algoritm mai bun poate fi obţinut dacă ţinem seama de rezultatul descoperit de Lucas în 1876:

$$cmmdc(f_m, f_n) = f_{cmmdc(m,n)}$$

Deci putem rezolva problema calculând un singur termen al şirului lui Fibonacci.

Există mai mulţi algoritmi de rezolvare a unei probleme date. Prin urmare, se impune o analiză a acestora, în scopul determinării eficienţei algoritmilor de rezolvare a problemei şi pe cât posibil a optimalităţii lor. Criteriile în funcţie de care vom stabili eficienţa unui algoritm sunt *complexitatea spaţiu* (memorie utilizată) şi *complexitatea timp* (numărul de operaţiilor elementare).

### 4.1.1 Complexitatea spaţiu

Prin *complexitate spaţiu* înţelegem dimensiunea spaţiului de memorie utilizat de program.

Un program necesită un spaţiu de memorie constant, independent de datele de intrare, pentru memorarea codului, a constantelor, a variabilelor şi a structurilor de date de dimensiune constantă alocate static şi un spaţiu de memorie variabil, a cărui dimensiune depinde de datele de intrare, constând din spaţiul necesar pentru structurile de date alocate dinamic, a căror dimensiune depinde de instanţa problemei de rezolvat şi din spaţiul de memorie necesar apelurilor de proceduri şi funcţii.

Progresele tehnologice fac ca importanţa criteriului *spaţiu de memorie* utilizat să scadă, prioritar devenind *criteriul timp*.

### 4.1.2 Complexitatea timp

Prin *complexitate timp* înţelegem timpul necesar execuţiei programului.

Înainte de a evalua timpul necesar execuţiei programului ar trebui să avem informaţii detaliate despre sistemul de calcul folosit.

Pentru a analiza teoretic algoritmul, vom presupune că se lucrează pe un calculator "clasic", în sensul că o singură instrucţiune este executată la un moment dat. Astfel, timpul necesar execuţiei programului depinde numai de numărul de operaţii elementare efectuate de algoritm.

Primul pas în analiza *complexităţii timp* a unui algoritm este determinarea operaţiilor elementare efectuate de algoritm şi a costurilor acestora.

Considerăm *operaţie elementară* orice operaţie al cărei timp de execuţie este independent de datele de intrare ale problemei.

Timpul necesar execuţiei unei operaţii elementare poate fi diferit de la o operaţie la alta, dar este fixat, deci putem spune că operaţiile elementare au timpul măginit superior de o constantă.

Fără a restrânge generalitatea, vom presupune că toate operaţiile elementare au acelaşi timp de execuţie, fiind astfel necesară doar evaluarea numărului de operaţii elementare, nu şi a timpului total de execuţie a acestora.

Analiza teoretică ignoră factorii care depind de calculator sau de limbajul de programare ales şi se axează doar pe determinarea *ordinului de mărime* a numărului de operaţi elementare.

Pentru a analiza timpul de execuţie se foloseşte deseori modelul Random Access Machine (RAM), care presupune: memoria constă într-un şir infinit de celule, fiecare celulă poate stoca cel mult o dată, fiecare celulă de memorie poate fi accesată într-o unitate de timp, instrucţiunile sunt executate secvenţial şi toate instrucţiunile de bază se execută într-o unitate de timp.

Scopul analizei teoretice a algoritmilor este de fapt determinarea unor funcţii care să limiteze superior, respectiv inferior comportarea în timp a algoritmului. Funcţiile depind de caracteristicile relevante ale datelor de intrare.

## 4.2 Notaţia asimptotică

### 4.2.1 Definire şi proprietăţi

**Definiţia 2** *Numim ordinul lui $f$, mulţimea de funcţii*

$$O(f) = \{t : \mathbb{N} \to \mathbb{R}_+ | \exists c > 0, \exists n_0 \in \mathbb{N} \text{ a.î. } t(n) \leq cf(n), \forall n > n_0\} \qquad (4.2.1)$$

Rezultă că $O(f)$ este mulţimea tuturor funcţiilor mărginite superior de un multiplu real pozitiv al lui $f$, pentru valori suficient de mari ale argumentului.

Dacă $t(n) \in O(f)$ vom spune că $t$ este *de ordinul* lui $f$ sau *în ordinul* lui $f$.

Fie un algoritm dat şi o funcţie $t : \mathbb{N} \to \mathbb{R}_+$, astfel încât o anumită implementare a algoritmului să necesite cel mult $t(n)$ unităţi de timp pentru a rezolva un caz de marime $n$.

Principiul invarianţei ne asigură că orice implementare a algoritmului necesită un timp în ordinul lui $t$. Mai mult, acest algoritm necesită un timp în ordinul lui $f$ pentru orice functie $f : \mathbb{N} \to \mathbb{R}_+$ pentru care $t \in O(f)$. În particular $t \in O(t)$. Vom căuta să găsim cea mai simplă funcţie astfel încât $t \in O(f)$.

Pentru calculul ordinului unei funcţii sunt utile următoarele proprietăţi:

**Proprietatea 1** $O(f) = O(g) \iff f \in O(g)$ *şi* $g \in O(f)$

**Proprietatea 2** $O(f) \subset O(g) \iff f \in O(g)$ *şi* $g \notin O(f)$

**Proprietatea 3** $O(f + g) = O(\max(f, g))$

Pentru calculul mulţimilor $O(f)$ şi $O(g)$ este utilă proprietatea următoare:

**Proprietatea 4** *Fie $f, g : \mathbb{N} \to \mathbb{R}^+$. Atunci*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow O(f) = O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g).$$

Reciproca nu este în general valabilă.

Fie de exemplu, $t(n) = n^2 + 3n + 2$, atunci

$$\lim_{n\to\infty} \frac{n^2 + 3n + 2}{n^2} = 1 \Longrightarrow O(n^2 + 3n + 2) = O(n^2)$$

$$\lim_{n\to\infty} \frac{n^2 + 3n + 2}{n^3} = 0 \Longrightarrow O(n^2 + 3n + 2) \subset O(n^3)$$

$$\lim_{n\to\infty}\frac{\ln(n)}{\sqrt{n}}=\lim_{n\to\infty}\frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=\lim_{n\to\infty}\frac{2}{\sqrt{n}}=0\Longrightarrow O(\ln(n))\subset O(\sqrt{n})$$

dar $O(\sqrt{n})\not\subset O(\ln(n)$

Dacă $p$ este un polinom de gradul $m$ în variabila $n$, atunci $O(p)=O(n^m)$.

Notaţia asimptotică defineşte o relaţie de ordine parţială între funcţii.

Pentru $f,g:\mathbb{N}\to\mathbb{R}^*$ notăm $f\prec g$ dacă $O(f)\subseteq O(g)$.

Această relaţie are proprietăţile corespunzătoare unei relaţii de ordine, adică:

a) reflexivitate: $f\prec f$

b) antisimetrie: dacă $f\prec g$ şi $g\prec f$ atunci $f=g$

c) tranzitivitate: $f\prec g$ şi $g\prec h$, implică $f\prec h$.

Dar nu este o relaţie de ordine! Există şi funcţii astfel încât $f\not\prec g$ ($f\notin O(g)$) şi $g\not\prec f$ ($g\notin O(f)$). De exemplu $f(n)=n$, $g(n)=n^{1+\sin(n)}$.

Putem defini şi o relaţie de echivalenţă: $f\equiv g$, dacă $O(f)=O(g)$. În mulţimea $O(f)$ putem înlocui orice funcţie cu o funcţie echivalentă cu ea. De exemplu: $\ln(n)\equiv\log(n)\equiv\log_2(n)$.

Notând cu $O(1)$ mulţimea funcţiilor mărginite superior de o constantă şi considerând $m\in N$ , $m\geq 2$, obţinem ierarhia:

$$O(1)\subset O(\log(n))\subset O(\sqrt{n})\subset O(n)\subset O(n\cdot log(n))\subset O(n^m)\subset O(2^n)\subset O(n!)$$

şi evident $O(n^2)\subset O(n^3)\subset ...\subset O(n^m)$ pentru $m\geq 4$.

Această ierarhie corespunde ierarhiei algoritmilor după criteriul performanţei. Pentru o problemă dată, dorim sa realizăm un algoritm cu un ordin situat cât mai în stânga în această ierarhie.

Notatia $O(f)$ este pentru a delimita superior timpul necesar unui algoritm.

Notăm $T_A(n)$ timpul necesar execuţiei algoritmului $A$.

Fie $f:\mathbb{N}\to\mathbb{R}_+^*$ o funcţie arbitrară. Spunem că *algoritmul este de ordinul lui* $f(n)$ (şi notăm $T_A(n)\in O(f(n))$), dacă şi numai dacă există $c>0$ şi $n_0\in\mathbb{N}$, astfel încât $T_A(n)\leq c\cdot f(n)$, $\forall n\geq n_0$.

De exemplu:

a) Dacă $T_A(n)=3n+2$, atunci $T_A(n)\in O(n)$, pentru că $3n+2\leq 4n$, $\forall n\geq 2$.

Mai general, dacă $T_A(n)=a\cdot n+b$, $a>0$, atunci $T_A(n)\in O(n)$ pentru că există $c=a+1>0$ şi $n_0=b\in\mathbb{N}$, astfel încât $a\cdot n+b\leq(a+1)\cdot n$, $\forall n\geq b$.

b) Dacă $T_A(n)=10n^2+4n+2$, atunci $T_A(n)\in O(n^2)$, pentru că $10n^2+4n+2\leq 11n^2$, $\forall n\geq 5$.

Mai general, dacă $TA(n)=an^2+bn+c$, $a>0$, atunci $TA(n)\in O(n^2)$, pentru că $an^2+bn+c\leq(a+1)n^2$, $\forall n\geq max(b,c)+1$.

c) Dacă $T_A(n)=6\cdot 2^n+n^2$, atunci $TA(n)\in O(2^n)$, pentru că $T_A(n)\leq 7\cdot 2^n$,$\forall n\geq 4$.

Dacă $T_A(n)=a_kn^k+a_{k-1}n^{k-1}+...+a_1n+a_0$, atunci $TA(n)\in O(n^k)$. Aceasta rezultă din: $T_A(n)=|T_A(n)|=|a_kn^k+a_{k-1}n^{k-1}+...+a_1n+a_0|\leq|a_k|n^k+|a_{k-1}|n^{k-1}+...+|a_1|n+|a_0|\leq(|a_k|+|a_{k-1}|+...+|a_1|+|a_0|)n^k$, $\forall n\geq 1$ şi alegând $c=|a_k|+|a_{k-1}|+...+|a_1|+|a_0|$ şi $n=1$ rezultă $T_A(n)\in O(n^k)$.

### 4.2.2 Clase de complexitate

Notaţia $O$ oferă o limită superioară a timpului de execuţie a unui algoritm.

Un algoritm cu $T_A(n) \in O(1)$ necesită un timp de execuţie constant. Un algoritm cu $T_A(n) \in O(n)$ se numeşte *liniar*. Dacă $T_A(n) \in O(n^2)$ algoritmul se numeşte *pătratic*, iar dacă $T_A(n) \in O(n^3)$, *cubic*. Un algoritm cu $T_A(n) \in O(n^k)$ se numeşte *polinomial*, iar dacă $T_A(n) \in O(2^n)$ algoritmul se numeşte *exponenţial*.

Tabelul următor ilustrează comportarea a cinci din cele mai importante funcţii de complexitate.

| $O(log(n))$ (logaritmic) | $O(n)$ (liniar) | $O(n.log(n))$ (log-liniar) | $O(n^2)$ (pătratic) | $O(n^3)$ cubic | $O(2^n)$ (exponenţial) |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 2 |
| 1 | 2 | 2 | 4 | 8 | 4 |
| 2 | 4 | 8 | 16 | 64 | 16 |
| 3 | 8 | 24 | 64 | 512 | 256 |
| 4 | 16 | 64 | 256 | 4096 | 65536 |
| 5 | 32 | 160 | 1024 | 32768 | 4294967296 |

Tabelul 4.1: Funcţii de complexitate

Dacă $T_A(n) \in O(2^n)$, pentru $n = 40$, pe un calculator care face $10^9$ de operaţii pe secundă, sunt necesare aproximativ 18 minute. Pentru $n = 50$, acelaşi program va rula 13 zile pe acest calculator, pentru $n = 60$, vor fi necesari peste 310 ani, iar pentru $n = 100$ aproximativ $4.10^{13}$ ani.

Utilitatea algoritmilor polinomiali de grad mare este de asemenea limitată. De exemplu, pentru $O(n^{10})$, pe un calculator care execută $10^9$ operaţii pe secundă sunt necesare 10 secunde pentru $n = 10$, aproximativ 3 ani pentru $n = 100$ şi circa $3.10^{13}$ ani pentru $n = 1000$.

Uneori este util să determinăm şi o limită inferioară pentru timpul de execuţie a unui algoritm. Notaţia matematică este $\Omega$.

**Definiţie:** Spunem că $T_A(n) \in \Omega(f(n))$ dacă şi numai dacă $\exists c > 0$ şi $n_0 \in \mathbb{N}$ astfel încât $T_A(n) \geq c \cdot f(n)$, $\forall n \geq n_0$.

De exemplu:

a) dacă $T_A(n) = 3n+2$, atunci $T_A(n) \in \Omega(n)$, pentru că $3n+2 \geq 3n$, $\forall n \geq 1$;

b) dacă $T_A(n) = 10n^2+4n+2$, atunci $T_A(n) \in \Omega(n)$, pentru că $10n^2+4n+2 \geq n2$, $\forall n \geq 1$;

c) dacă $T_A(n) = 6 \cdot 2^n + n^2$, atunci $T_A(n) \in \Omega(2^n)$, pentru că $6 \cdot 2^n + n^2 \geq 2^n$, $\forall n \geq 1$.

Există funcţii $f$ care constituie atât o limită superioară cât şi o limită inferioară a timpului de execuţie a algoritmului. De exemplu, dacă $T_A(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0$, $a_k > 0$ atunci $T_A(n) \in \Omega(n^k)$.

**Definiţie :** Spunem că $T_A(n) \in \Theta(f(n))$ dacă şi numai dacă $\exists c_1, c_2 > 0$ şi $n_0 \in \mathbb{N}$ astfel încât $c_1 \cdot f(n) \leq T_A(n) \leq c_2 \cdot f(n)$, $\forall n \geq n_0$.

În acest caz $f(n)$ constituie atât o limită inferioară cât şi o limită superioară pentru timpul de execuţie a algoritmului. Din acest motiv $\Theta$ se poate numi *ordin exact*. Se poate arăta uşor că $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$. De asemenea, dacă $T_A(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0$, $a_k > 0$ atunci $T_A(n) \in \Theta(n^k)$.

### 4.2.3 Cazul mediu şi cazul cel mai defavorabil

Am arătat că timpul de execuţie al unui algoritm este direct proporţional cu numărul de operaţii elementare şi am stabilit o notaţie asimptotică pentru timpul de execuţie. Totuşi, numărul de operaţii elementare efectuate de algoritm poate varia considerabil pentru diferite seturi de date de intrare.

Determinarea *complexităţii timp* a algoritmului ca o funcţie de caracteristicile datelor de intrare este o sarcină uşoară doar pentru algoritmi relativ simpli, dar în general problema este dificilă şi din această cauză analizăm complexitatea algoritmilor *în medie* sau *în cazul cel mai defavorabil*.

*Complexitatea în cazul cel mai defavorabil* este numărul maxim de operaţii elementare efectuate de algoritm.

Dar chiar dacă este cunoscut cazul cel mai defavorabil, datele utilizate efectiv în practică pot conduce la timpi de execuţie mult mai mici. Numeroşi algoritmi foarte utili au o comportare convenabilă în practică, dar foarte proastă în cazul cel mai defavorabil.

Cel mai cunoscut exemplu este algoritmul de sortare rapidă (quicksort) care are complexitatea în cazul cel mai defavorabil de $O(n^2)$, dar pentru datele întâlnite în practică funcţionează în $O(n \cdot \log n)$.

Determinarea *complexităţii în medie* necesită cunoaşterea repartiţiei probabilistice a datelor de intrare şi din acest motiv analiza complexităţii în medie este mai dificil de realizat. Pentru cazuri simple, de exemplu un algoritm de sortare care acţioneaza asupra unui tablou cu $n$ componente întregi aleatoare sau un algoritm geometric pe o mulţime de $N$ puncte în plan de coordonate aleatoare cuprinse în intervalul $[0, 1]$, putem caracteriza exact datele de intrare.

Dacă notăm:

- $D$ - spaţiul datelor de intrare
- $p(d)$ - probabilitatea apariţiei datei $d \in D$ la intrarea algoritmului
- $T_A(d)$ - numărul de operaţii elementare efectuate de algoritm pentru $d \in D$

atunci *complexitatea medie* este

$$\sum_{d \in D} p(d) \cdot T_A(d).$$

### 4.2.4 Analiza asimptotică a structurilor fundamentale

Considerăm problema determinării ordinului de complexitate în cazul cel mai defavorabil pentru structurile algoritmice: secvenţială, alternativă şi repetitivă.

Presupunem că structura secvenţială este constituită din prelucrările $A_1$, $A_2$, ..., $A_k$ şi fiecare dintre acestea are ordinul de complexitate $O(g_i(n)), 1 \leq i \leq n$. Atunci structura va avea ordinul de complexitate $O(\max\{g_1(n), ..., g_k(n)\})$.

Dacă condiţia unei structuri alternative are cost constant iar prelucrările celor două variante au ordinele de complexitate $O(g_1(n))$ respectiv $O(g_2(n))$ atunci costul structurii alternative va fi $O(\max\{g_1(n), g_2(n)\})$.

În cazul unei structuri repetitive pentru a determina ordinul de complexitate în cazul cel mai defavorabil se consideră numărul maxim de iteraţii. Dacă acesta este $n$ iar în corpul ciclului prelucrările sunt de cost constant atunci se obţine ordinul $O(n)$.

## 4.3 Exemple

### 4.3.1 Calcularea maximului

Fiind date $n$ elemente $a_1, a_2, ..., a_n$, să se calculeze $\max\{a_1, a_2, ..., a_n\}$.

```
max = a[1];
for i = 2 to n do
   if a[i] > max
      then max = a[i];
```

Vom estima timpul de execuţie al algoritmului în funcţie de $n$, numărul de date de intrare. Fiecare iteraţie a ciclului **for** o vom considera operaţie elementară. Deci complexitatea algoritmului este $O(n)$, atât în medie cât şi în cazul cel mai defavorabil.

### 4.3.2 Sortarea prin selecţia maximului

Sortăm crescător vectorul $a$, care are $n$ componente.

```
for j=n,n-1,...,2
{
  max=a[1];
  pozmax=1;
  for i=2,3,...,j
  {
    if a[i]>max { a[i]=max; pozmax=i; }
```

```
      a[pozmax]=a[j];
      a[j]=max;
    }
  }
```

Estimăm complexitatea algoritmului în funcţie de $n$, dimensiunea vectorului. La fiecare iteraţie a ciclului for exterior este calculat $\max\{a_1, a_2, ..., a_j\}$ şi plasat pe poziţia $j$, elementele de la $j+1$ la $n$ fiind deja plasate pe poziţiile lor definitive.

Conform exemplului anterior, pentru a calcula $\max\{a_1, a_2, ..., a_j\}$ sunt necesare $j-1$ operaţii elementare, în total $1+2+...+(n-1) = n(n-1)/2$. Deci complexitatea algoritmului este de $O(n^2)$. Să observăm că timpul de execuţie este independent de ordinea iniţială a elementelor vectorului.

### 4.3.3 Sortarea prin inserţie

Este o metodă de asemenea simplă, pe care o utilizăm adesea când ordonăm cărţile la jocuri de cărţi.

```
for i=2,3,...,n
{
  val=a[i];
  poz=i;
  while a[poz-1]>val
  {
    a[poz]=a[poz-1];
    poz=poz-1;
  }
  a[poz]=val;
}
```

Analizăm algoritmul în funcţie de $n$, dimensiunea vectorului ce urmează a fi sortat. La fiecare iteraţie a ciclului for elementele $a_1, a_2, ..., a_{i-1}$ sunt deja ordonate şi trebuie să inserăm valorea $a[i]$ pe poziţia corectă în şirul ordonat. În cazul cel mai defavorabil, când vectorul este iniţial ordonat descrescător, fiecare element $a[i]$ va fi plasat pe prima poziţie, deci ciclul while se execută de $i-1$ ori. Considerând drept operaţie elementară comparaţia $a[poz-1] > val$ urmată de deplasarea elementului de pe poziţia $poz-1$, vom avea în cazul cel mai defavorabil $1+2+...+(n-1) = n(n-1)/2$ operaţii elementare, deci complexitatea algoritmului este de $O(n^2)$.

Să analizăm comportarea algoritmului în medie. Considerăm că elementele vectorului sunt distincte şi că orice permutare a lor are aceeaşi probabilitate de apariţie. Atunci probabilitatea ca valoarea $a_i$ să fie plasată pe poziţia $k$ în şirul $a_1, a_2, ..., a_i$, $k \in \{1, 2, ...., i\}$ este $1/i$. Pentru $i$ fixat, numărul mediu de operaţii elementare este:

$$\sum_{k=1}^{i} \frac{1}{i} \cdot (k-1) = \frac{1}{i} \cdot \sum_{k=1}^{i} (k-1) = \frac{1}{i} \left( \frac{i(i+1)}{2} - i \right) = \frac{i+1}{2} - 1 = \frac{i-1}{2}$$

Pentru a sorta cele $n$ elemente sunt necesare

$$\sum_{i=2}^{n} \frac{i-1}{2} = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1 - (n-1)\right) = \frac{n}{2}\left(\frac{(n+1)}{2} - 1\right) = \frac{n(n-1)}{4}$$

operaţii elementare. Deci complexitatea algoritmului în medie este tot $O(n^2)$.

### 4.3.4 Sortarea rapidă (quicksort)

Acest algoritm a fost elaborat de C.A.R. Hoare în 1960 şi este unul dintre cei mai utilizaţi algoritmi de sortare.

```
void quicksort(int st, int dr)
{
  int m;
  if st<dr
  {
    m=divide(st, dr);
    quicksort(st, m-1);
    quicksort(m+1, dr);
  }
}
```

Iniţial apelăm `quicksort(1,n)`.

Funcţia **divide** are rolul de aplasa primul element (a[st]) pe poziţia sa corectă în şirul ordonat. În stânga sa se vor găsi numai elemente mai mici, iar în dreapta numai elemente mai mari decât el.

```
int divide(int st, int dr)
{
  int i, j, val;
  val=a[st];
  i=st; j=dr;
  while(i<j)
  {
    while((i<j) && (a[j] >= val)) j=j-1;
    a[i]=a[j];
    while((i<j) && (a[i] <= val)) i=i+1;
    a[j]=a[i];
  }
  a[i]=val;
  return i;
}
```

Observaţie : Vectorul $a$ este considerat variabilă globală.

În cazul cel mai defavorabil, când vectorul $a$ era iniţial ordonat, se fac $n-1$ apeluri succesive ale procedurii `quicksort`, cu parametrii $(1,n)$, $(1,n-1)$, ..., $(1,2)$ (dacă vectorul a era iniţial ordonat descrescător) sau $(1,n)$, $(2,n)$, ..., $(n-1,n)$ (dacă vectorul $a$ era ordonat crescător).

La fiecare apel al procedurii `quicksort` este apelată funcţia `divide(1,i)` (respectiv `divide(i, n)`) care efectuează $i-1$, (respectiv $n-i-1$) operaţii elementare. În total numărul de operaţii elementare este $(n-1)+(n-2)+...+1 = n(n-1)/2$. Complexitatea algoritmului în cazul cel mai defavorabil este de $O(n^2)$.

Să analizăm comportarea algoritmului în medie. Vom considerăm că orice permutare a elementelor vectorului are aceeaşi probabilitate de apariţie şi notăm cu $T_n$ numărul de operaţii elementare efectuate pentru a sorta $n$ elemente.

Probabilitatea ca un element al vectorului să fie plasat pe poziţia $k$ în vectorul ordonat, este de $1/n$.

$$T_n = \begin{cases} 0, & \text{dacă } n = 0 \text{ sau } n = 1 \\ \frac{1}{n}\sum_{k=1}^{n}(T_{k-1}+T_{n-k})+(n-1), & \text{dacă } n > 1 \end{cases}$$

(pentru a ordona crescător $n$ elemente, determinăm poziţia $k$ în vectorul ordonat a primului element, ceea ce necesită $n-1$ operaţii elementare, sortăm elementele din stânga, ceea ce necesită $T_{k-1}$ operaţii elementare, apoi cele din dreapta, necesitând $T_{n-k}$ operaţii elementare).

Problema se reduce la a rezolva relaţia de recurenţă de mai sus. Mai întâi observăm că

$$T_0 + T_1 + ... + T_{n-1} = Tn - 1 + ... + T_1 + T_0.$$

Deci,

$$T_n = n - 1 + \frac{2}{n}\sum_{k=1}^{n} T_{k-1}$$

Înmulţim ambii membri ai acestei relaţii cu $n$. Obţinem:

$$nT_n = n(n-1) + 2\sum_{k=1}^{n} T_{k-1}$$

Scăzând din această relaţie, relaţia obţinută pentru $n-1$, adică

$$(n-1)T_{n-1} = (n-1)(n-2) + 2\sum_{k=1}^{n-1} T_{k-1}$$

obţinem

$$nT_n - (n-1)T_{n-1} = n(n-1) - (n-1)(n-2) + 2T_{n-1}$$

de unde rezultă

$$nT_n = 2(n-1) + (n+1)T_{n-1}$$

Împărţind ambii membri cu $n(n+1)$ obţinem

$$\frac{T_n}{n+1} = \frac{T_{n-1}}{n} + \frac{2(n-1)}{n(n+1)} = \frac{T_{n-2}}{n-1} + \frac{2(n-1)}{n(n+1)} + \frac{2(n-2)}{(n-1)n} = ... = \frac{T_2}{3} + 2\sum_{k=3}^{n} \frac{k-1}{k(k+1)}$$

Deci

$$\frac{T_n}{n+1} = \frac{T_2}{3} + 2\sum_{k=3}^{n} \left( \frac{1}{k+1} - \frac{1}{k} + \frac{1}{k+1} \right) = \frac{T_2}{3} + \frac{2}{n+1} + 2\sum_{k=3}^{n} \frac{1}{k} \approx 2\sum_{k=1}^{n} \frac{1}{k} \approx 2\ln n$$

Deci, în medie, complexitatea algoritmului este de $O(n \log n)$.

### 4.3.5 Problema celebrităţii

Numim *celebritate* o persoană care este cunoscută de toată lumea, dar nu cunoaşte pe nimeni. Se pune problema de a identifica o celebritate, dacă există, într-un grup de $n$ persoane pentru care relaţiile dintre persoane sunt cunoscute.

Putem reformula problema în limbaj de grafuri astfel: fiind dat un digraf cu $n$ vârfuri, verificaţi dacă există un vârf cu gradul exterior 0 şi gradul interior $n-1$.

Reprezentăm graful asociat problemei prin matricea de adiacenţă $a_{n\times n}$

$$a_{i,j} = \begin{cases} 1, & \text{dacă persoana } i \text{ cunoaste persoana } j; \\ 0, & \text{altfel.} \end{cases}$$

O primă soluţie ar fi să calculăm pentru fiecare persoană $p$ din grup numărul de persoane pe care $p$ le cunoaşte ($out$) şi numărul de persoane care cunosc persoana $p$ ($in$). Cu alte cuvinte, pentru fiecare vârf din digraf calculăm gradul interior şi gradul exterior. Dacă găsim o persoană pentru care $out = 0$ şi $in = n-1$, aceasta va fi celebritatea căutată.

```
celebritate=0;
for p=1,2,...,n
{
  in=0; out=0;
  for j=1,2,...,n
  {
    in=in+a[j][p];
    out=out+a[p][j];
  }
  if (in=n-1) and (out = 0) celebritate=p;
}
if celebritate=0 writeln('Nu exista celebritati !')
   else writeln(p, ' este o celebritate.');
```

Se poate observa cu uşurinţă că algoritmul este de $O(n^2)$. Putem îmbunătăţi algoritmul făcând observaţia că atunci când testăm relaţiile dintre persoanele $x$ şi $y$ apar următoarele posibilităţii:

$a[x, y] = 0$ şi în acest caz $y$ nu are nici o şansă să fie celebritate, sau

$a[x, y] = 1$ şi în acest caz $x$ nu poate fi celebritate.

Deci la un test eliminăm o persoană care nu are şanse să fie celebritate.

Făcând succesiv $n - 1$ teste, în final vom avea o singură persoană candidat la celebritate. Rămâne să calculăm numărul de persoane cunoscute şi numărul de persoane care îl cunosc pe acest candidat, singura celebritate posibilă.

```
candidat=1;
for i=2,n
  if a[candidat][i]=1 candidat=i;
out=0;
in=0;
for i=1,n
{
  in=in+a[i][candidat];
  out=out+a[candidat][i];
}
if (out=0) and (in=n-1) write(candidat, ' este o celebritate .')
  else write('Nu exista celebritati.');
```

În acest caz algoritmul a devenit liniar.

## 4.4 Probleme

### 4.4.1 Probleme rezolvate

**Problema 1** *Care afirmaţii sunt adevarate:*

*a)* $n^2 \in O(n^3)$

*b)* $n^3 \in O(n^2)$

*c)* $2^{n+1} \in O(2^n)$

*d)* $(n+1)! \in O(n!)$

*e)* $\forall f : \mathbb{N} \to \mathbb{R}^*, f \in O(n) \Longrightarrow f^2 \in O(n^2)$

*f)* $\forall f : \mathbb{N} \to \mathbb{R}^*, f \in O(n) \Longrightarrow 2^f \in O(2^n)$

Rezolvare:

a) Afirmaţia este adevarată pentru că: $\lim_{n\to\infty} \frac{n^2}{n^3} = 0 \Longrightarrow n^2 \in O(n^3)$.

b) Afirmaţia este falsă pentru că: $\lim_{n\to\infty} \frac{n^3}{n^2} = \infty$

c) Afirmaţia este adevarată pentru că: $\lim_{n\to\infty} \frac{2^{n+1}}{2^n} = 2 \Longrightarrow O(2^{n+1}) = O(2^n)$.

d) Afirmaţia este falsă pentru că: $\lim_{n\to\infty} \frac{(n+1)!}{n!} = \lim_{n\to\infty} \frac{n+1}{1} = \infty$

e) Afirmaţia este adevarată pentru că: $f \in O(n) \Longrightarrow \exists c > 0$ şi $\exists n_0 \in \mathbb{N}$ astfel încât $f(n) < c \cdot n$, $\forall n > n_0$. Rezultă că $\exists c_1 = c^2$ astfel încâ $f^2(n) < c_1 \cdot n^2$, $\forall n > n_0$, deci $f^2 \in O(n^2)$.

e) Afirmaţia este adevarată pentru că: $f \in O(n) \Longrightarrow \exists c > 0$ şi $\exists n_0 \in \mathbb{N}$ astfel încât $f(n) < c \cdot n$, $\forall n > n_0$. Rezultă că $\exists c_1 = 2^c$ astfel încâ $2^{f(n)} < 2^{c \cdot n} = 2^c \cdot 2^n = c_1 \cdot 2^n$, $\forall n > n_0$, deci $2^f \in O(2^n)$.

**Problema 2** *Arătaţi că* $\log n \in O(\sqrt{n})$ *dar* $\sqrt{n} \notin O(\log n)$.

Indicaţie: Prelungim domeniile funcţiilor pe $\mathbb{R}^+$, pe care sunt derivabile, şi aplicăm relula lui L'Hôspital pentru $\log n/\sqrt{n}$.

**Problema 3** *Demonstraţi următoarele afirmaţii:*

$$
\begin{aligned}
&i) \quad \log_a \in \Theta(\log_b n), \text{ pentru oricare } a, b > 1 \\
&ii) \quad \sum_{i=1}^{n} i^k \in \Theta(n_{k+1}), \text{ pentru oricare } k \in \mathbb{N} \\
&iii) \quad \sum_{i=1}^{n} \frac{1}{i} \in \Theta(n \log n) \\
&iv) \quad \log n! \in \Theta(n \log n)
\end{aligned}
$$

Indicaţii: La punctul *iii*) se ţine cont de relaţia

$$\sum_{i=1}^{\infty} \frac{1}{i} \approx \gamma + \ln n$$

unde $\gamma \approx 0.5772$ este constanta lui Euler.

La punctul *iv*) din $n! < n^n$, rezultă $\log n! < n \log n$, deci $\log n! \in O(n \log n)$. Trebuie să găsim şi o margine inferioară. Pentru $0 \leq i \leq n-1$ este adevărată relaţia

$$(n-i)(i+1) \geq n$$

Deoarece

$$(n!)^2 = (n \cdot 1)((n-1) \cdot 2)((n-2) \cdot 3)...(2 \cdot (n-1))(1 \cdot n) \geq n^n$$

rezultă $2 \log n! \geq n \log n$, adică $\log n! \geq 0.5n \log n$, deci $\log n! \in \Omega(n \log n)$.

Relaţia se poate demonstra şi folosind aproximarea lui Stirling

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + \Theta(1/n))$$

### 4.4.2 Probleme propuse

1. Arătaţi că:
a) $n^3 + 10^6 n \in \Theta(n^3)$
b) $n^{2^n} + 6 \cdot 2^n \in \Theta()(n^{2^n})$
c) $2n^2 + n \log n \in \Theta(n^2)$
d) $n^k + n + n^k \log n \in \Theta(n^k \log n)$, $k \geq 1$
e) $\log_a n \in \Theta(\log_b n)$, $a, b > 0$, $a \neq 1, b \neq 1$.

2. Pentru oricare doua functii $f, g : \mathbb{N} \to \mathbb{R}^*$ demonstraţi că

$$O(f + g) = O(\max(f, g)) \tag{4.4.1}$$

unde suma şi maximul se iau punctual.

3. Fie $f, g : \mathbb{N} \to \mathbb{R}^+$ Demonstraţi că:

$$i) \lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow O(f) = O(g), \qquad ii) \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g)$$

Observaţie: Implicaţiile inverse nu sunt în general adevărate, deoarece se poate întampla ca limitele să nu existe.

4. Demonstraţi prin inducţie că pentru a determina maximul a $n$ numere sunt necesare $n - 1$ comparaţii.

5. Care este timpul de execuţie a algoritmului `quicksort` pentru un vector cu $n$ componente egale?

6. Să considerăm următorul algoritm de sortare a unui vector a cu $n$ componente:

```
do
{
  ok=true;
  for i=1,n-1
    if a[i]>a[i+1] { aux=a[i]; a[i]=a[i+1]; a[i+1]= aux; }
  ok=false;
} while !ok;
```

Analizaţi algoritmul în medie şi în cazul cel mai defavorabil.

7. Analizaţi complexitatea algoritmului de interclasare a doi vectori ordonaţi, $a$ cu $n$ componente, respectiv $b$ cu $m$ componente :

```
i=1; j=1; k=0;
while (i <=  n) and (j <=  m)
{
  k=k+1;
  if a[i] < b[j] { c[k]=a[i]; i=i+1; }
    else { c[k]=b[j]; j=j+1; }
```

```
}
for t=i,n { k=k+1; c[k]=a[t]; }
for t=j,m { k=k+1; c[k]=b[t]; }
```

8. Fiind dat $a$, un vector cu $n$ componente distincte, verificaţi dacă o valoare dată $x$ se găseşte sau nu în vector. Evaluaţi complexitatea algoritmului în cazul cel mai defavorabil şi în medie.

9. Se dă $a$ un vector cu $n$ componente. Scrieţi un algoritm liniar care să determine cea mai lungă secvenţă de elemente consecutive de valori egale.

10. Fie $T$ un text. Verificaţi în timp liniar dacă un text dat $T'$ este o permutare circulară a lui $T$.

11. Fie $X = (x_1, x_2, ..., x_n)$ o secvenţă de numere întregi. Fiind dat $x$, vom numi multiplicitate a lui $x$ în $X$ numărul de apariţii ale lui $x$ în $X$. Un element se numeşte majoritar dacă multiplicitatea sa este mai mare decât $n/2$. Descrieţi un algoritm liniar care să determine elementul majoritar dintr-un şir, dacă un astfel de element există.

12. Fie $\{a_1, a_2, ..., a_n\}$ şi $\{b_1, b_2, ..., b_m\}$, două mulţimi de numere întregi, nenule ($m < n$). Să se determine $\{x_1, x_2, ..., x_m\}$, o submulţime a mulţimii $\{a_1, a_2, ..., a_n\}$ pentru care funcţia $f(x_1, x_2, ..., x_m) = a_1x_1 + a_2x_2 + ... + a_nx_m$ ia valoare maximă, prin doi algoritmi de complexitate diferită.

# Capitolul 5

# Recursivitate

Definiţiile prin recurenţă sunt destul de curente în matematică: progresia aritmetică, progresia geometrică, şirul lui Fibonacci, limite de şiruri, etc.

## 5.1 Funcţii recursive

### 5.1.1 Funcţii numerice

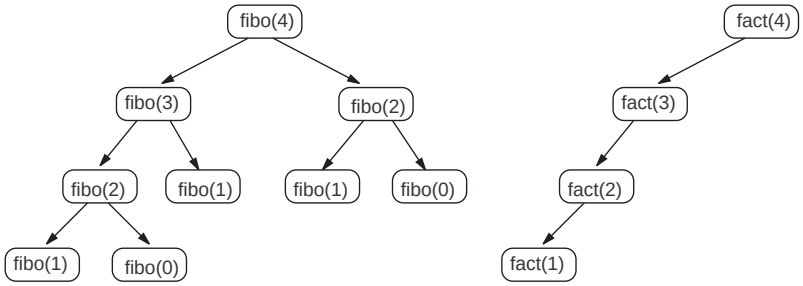
Pentru calculul termenilor şirului lui Fibonacci, a transcriere literală a formulei este următoarea:

```
static int fib(int n) {
  if (n <= 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

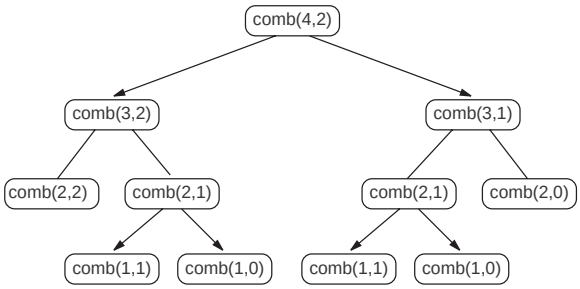
`fib` este o funcţie care utilizează propriul nume în definiţia proprie. De asemenea, dacă argumentul $n$ este mai mic decât 1 returnează valoarea 1 iar în caz contrar returnează $fib(n-1) + fib(n-2)$.

În Java este posibil, ca de altfel în multe alte limbaje de programare (Fortran, Pascal, C, etc), să definim astfel de funcţii *recursive*. Dealtfel, toate şirurile definite prin recurenţă se scriu în această manieră în Java, cum se poate observa din următoarele două exemple numerice: factorialul şi triunghiul lui Pascal.

```
static int fact(int n) {
  if (n != 1)
    return 1;
  else
    return n * fact (n1);
}
```




```
static int comb(int n, int p) {
  if ((p == 0) || (p == n))
    return 1;
  else
    return comb(n-1, p) + comb(n-1, p-1);
}
```




Ne putem întreba cum efectuează Java calculul funcţiilor recursive. Putem să răspundem prin urmărirea calculelor în cazul calculului lui $fibo(4)$. Reamintim că argumentele sunt transmise prin valoare în acest caz, iar un apel de funcţie constă în evaluarea argumentului, apoi lansarea în execuţie a funcţiei cu valoarea

argumentului. Deci

$$\begin{aligned}
fibo(4) &\rightarrow fibo(3) + fibo(2) \\
&\rightarrow (fibo(2) + fibo(1)) + fibo(2) \\
&\rightarrow ((fibo(1) + fibo(1)) + fibo(1)) + fibo(2) \\
&\rightarrow ((1 + fibo(1)) + fibo(1)) + fibo(2) \\
&\rightarrow ((1 + 1) + fibo(1)) + fibo(2) \\
&\rightarrow (2 + fibo(1)) + fibo(2) \\
&\rightarrow (2 + 1) + fibo(2) \\
&\rightarrow 3 + fibo(2) \\
&\rightarrow 3 + (fibo(1) + fibo(1)) \\
&\rightarrow 3 + (1 + fibo(1)) \\
&\rightarrow 3 + (1 + 1) \\
&\rightarrow 3 + 2 \\
&\rightarrow 5
\end{aligned}$$

Există deci un număr semnificativ de apeluri succesive ale funcţiei $fib$ (9 apeluri pentru calculul lui $fibo(4)$). Să notăm prin $R_n$ numărul apelurilor funcţiei $fibo$ pentru calculul lui $fibo(n)$. Evident $R_0 = R_1 = 1$, şi $R_n = 1 + R_{n-1} + R_{n-2}$ pentru $n > 1$. Punând $R'_n = R_n + 1$, obţinem că $R'_n = R'_{n-1} + R'_{n-2}$ pentru $n > 1$, şi $R'_1 = R'_0 = 2$. Rezultă $R'_n = 2 \cdot fibo(n)$ şi de aici obţinem că $R_n = 2 \cdot fibo(n) - 1$. Numărul de apeluri recursive este foarte mare! Există o metodă iterativă simplă care permite calculul lui $fibo(n)$ mult mai repede.

$$\begin{pmatrix} fibo(n) \\ fibo(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} fibo(n-1) \\ fibo(n-2) \end{pmatrix} = ... = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \times \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} u0 \\ v0 \end{pmatrix}$$

```
static int fibo(int n) {
  int u, v;
  int u0, v0;
  int i;
  u = 1; v = 1;
  for (i = 2; i <= n; ++i) {
    u0 = u; v0 = v;
    u = u0 + v0;
    v = v0;
  }
  return u;
}
```

Se poate calcula şi mai repede folosind ultima formă şi calculând puterea matricei ...

Pentru a rezuma, o regulă bună este să nu încercăm să intrăm în meandrele detaliilor apelurilor recursive pentru a înţelege sensul unei funcţii recursive. În general este sufucient să înţelegem sintetic funcţia. Funcţia lui Fibonacci este un caz particular în care calculul recursiv este foarte lung. Cam la fel se întâmplă (dacă nu chiar mai rău!) şi cu triunghiul lui Pascal. Dar nu aceasta este situaţia în general. Nu numai că scrierea recursivă se poate dovedi eficace, dar ea este totdeauna naturală şi deci cea mai estetică. Ea nu face decât să respecte definiţia matematică prin recurenţă. Este o metodă de programare foarte puternică.

### 5.1.2 Funcţia lui Ackerman

Şirul lui Fibonacci are o creştere exponenţială. Există funcţii recursive care au o creştere mult mai rapidă. Prototipul este funcţia lui Ackerman. În loc să definim matematic această funcţie, este de asemenea simplu să dăm definiţia recursivă în Java.

```
static int ack(int m, int n) {
  if (m == 0)
    return n+1;
  else
    if (n == 0)
      return ack (m-1, 1);
    else
      return ack(m-1, ack(m, n-1));
}
```

Se poate verifica că $ack(0,n) = n+1$, $ack(1,n) = n+2$, $ack(2,n) \approx 2n$, $ack(3,n) \approx 2^n$, $ack(5,1) \approx ack(4,4) \approx 2^{65536} > 10^{80}$, adică numărul atomilor din univers [11].

### 5.1.3 Recursii imbricate

Funcţia lui Ackerman conţine două apeluri recursive imbricate ceea ce determină o creştere rapidă. Un alt exemplu este *"funcţia 91"* a lui MacCarty [11]:

```
static int f(int n) {
  if (n > 100)
    return n-10;
  else
    return f(f(n+11));
}
```

Pentru această funcţie, calculul lui $f(96)$ dă

$$f(96) = f(f(107)) = f(97) = ... = f(100) = f(f(111)) = f(101) = 91.$$

Se poate arăta că această funcţie va returna 91 dacă $n \leq 100$ şi $n - 10$ dacă $n > 100$. Această funcţie anecdotică, care foloseşte recursivitatea imbricată, este interesantă pentru cŭu este evident că o astfel de definiţie dă dă acelaşi rezultat.

Un alt exemplu este funcţia lui Morris [11] care are următoarea formă:

```
static int g(int m, int n) {
  if (m == 0)
    return 1;
  else
    return g(m-1, g(m, n));
}
```

Ce valoare are $g(1,0)$? Efectul acestui apel de funcţie se poate observa din definiţia ei: $g(1,0) = g(0, g(1,0))$. Se declanşează la nesfârşit apelul $g(1,0)$. Deci, calculul nu se va termina niciodată!

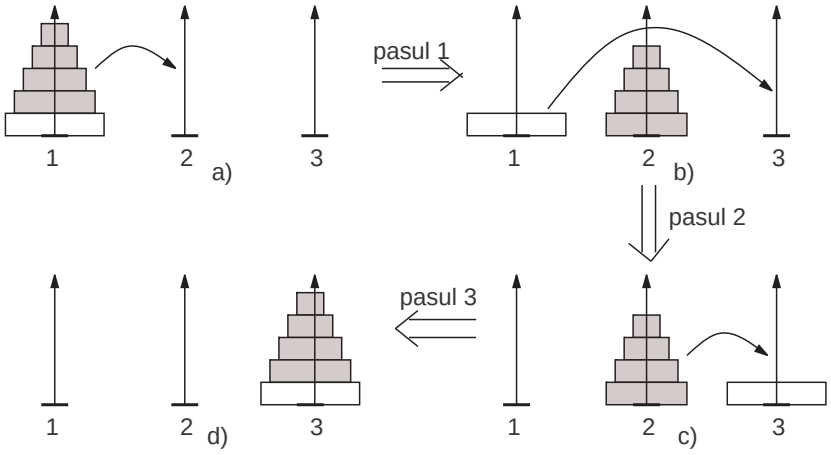
## 5.2 Proceduri recursive

Procedurile, la fel ca şi funcţiile, pot fi recursive şi pot suporta apeluri recursive. Exemplul clasic este cel al turnurilor din Hanoi. Pe 3 tije din faţa noastră, numerotate 1, 2 şi 3 de la stânga la dreapta, sunt $n$ discuri de dimensiuni diferite plasate pe tija 1 formând un con cu discul cel mai mare la bază şi cel mai mic în vârf. Se doreşte mutarea discurilor pe tija 3, mutând numai câte un singur disc şi neplasând niciodată un disc mai mare peste unul mai mic. Un raţionament recursiv permite scrierea soluţiei în câteva rânduri. Dacă $n \leq 1$, problema este trivială. Presupunem problema rezolvată pentru mutarea a $n - 1$ discuri de pe tija $i$ pe tija $j$ ($1 \leq i, j \leq 3$). Atunci, există o soluţie foarte simplă pentru mutarea celor $n$ discuri de pe tija $i$ pe tija $j$:

1. se mută primele $n-1$ discuri (cele mai mici) de pe tija $i$ pe tija $k = 6-i-j$,
2. se mută cel mai mare disc de pe tija $i$ pe tija $j$,
3. se mută cele $n - 1$ discuri de pe tija $k$ pe tija $j$.

```
static void hanoi(int n, int i, int j) {
  if (n > 0) {
    hanoi (n-1, i, 6-(i+j));
    System.out.println (i + " -> " + j);
    hanoi (n-1, 6-(i+j), j);
  }
}
```

Aceste câteva linii de program arată foarte bine cum generalizând problema, adică mutarea de pe oricare tijă $i$ pe oricare tijă$j$, un program recursiv de câteva linii poate rezolva o problemă apriori complicată. Aceasta este forţa recursivităţii şi a raţionamentului prin recurenţă.



# Capitolul 6

# Analiza algoritmilor recursivi

Am văzut în capitolul precedent cât de puternică şi utilă este recursivitatea în elaborarea unui algoritm. Cel mai important câştig al exprimării recursive este faptul că ea este naturală şi compactă.

Pe de altă parte, apelurile recursive trebuie folosite cu discernământ, deoarece solicită şi ele resursele calculatorului (timp si memorie).

Analiza unui algoritm recursiv implică rezolvarea unui sistem de recurenţe. Vom vedea în continuare cum pot fi rezolvate astfel de recurenţe.

## 6.1 Relaţii de recurenţă

O ecuaţie în care necunoscutele sunt termenii $x_n, x_{n+1}, ...x_{n+k}$ ai unui şir de numere se numeşte *relaţie de recurenţă de ordinul k*. Această ecuaţie poate fi satisfăcută de o infinitate de şiruri. Ca să putem rezolva ecuaţia (relaţia de recurenţă) mai avem nevoie şi de condiţii iniţiale, adică de valorile termenilor $x_0, x_1, ..., x_{k-1}$. De exemplu relaţia de recurenţă

$$(n+2)C_{n+1} = (4n+2)C_n, \text{ pentru } n \geq 0, \ C_0 = 1$$

este de ordinul 1.

Dacă un şir $x_n$ de numere satisface o formulă de forma

$$a_0x_n + a_1x_{n+1} + ... + a_kx_{n+k} = 0, k \geq 1, a_i \in \mathbb{R}, a_0, a_k \neq 0 \tag{6.1.1}$$

atunci ea se numeşte *relaţie de recurenţă de ordinul k cu coeficienţi constanţi*. Coeficienţii sunt constanţi în sensul că nu depind de valorile şirului $x_n$.

O astfel de formulă este de exemplu $F_{n+2} = F_{n+1} + F_n$, $F_0 = 0$, $F_1 = 1$, adică relaţia de recurenţă care defineşte şirul numerelor lui Fibonacci. Ea este o relaţie de recurenţă de ordinul 2 cu coeficienţi constanţi.

### 6.1.1 Ecuaţia caracteristică

Găsirea expresiei lui $x_n$ care să satisfacă relaţia de recurenţă se numeşte *rezolvarea relaţiei de recurenţă*. Făcând substituţia

$$x_n = r^n$$

obţinem următoarea ecuaţie, numită *ecuaţie caracteristică*:

$$a_0 + a_1 r + a_2 r^2 + ... + a_k r^k = 0 \tag{6.1.2}$$

### 6.1.2 Soluţia generală

Soluţia generală a relaţiei de recurenţă omogenă de ordinul $k$ cu coeficienţi constanţi este de forma

$$x_n = \sum_{i=1}^{k} c_i x_n^{(i)} \tag{6.1.3}$$

unde $\left\{x_n^{(i)} | i \in \{1, 2, ..., k\}\right\}$ sunt soluţii liniar independente ale relaţiei de recurenţă (se mai numesc şi *sistem fundamental de soluţii*). Pentru determinarea acestor soluţii distingem următoarele cazuri:

• **Ecuaţia caracteristică admite rădăcini reale şi distincte**

Dacă $r_1, r_2, ..., r_k$ sunt rădăcini reale ale ecuaţiei caracteristice, atunci $r_i^n$ sunt soluţii ale relaţiei de recurenţă.

Într-adevăr, introducând expresiile $r_i^n$ în relaţia de recurenţă, obţinem:

$$a_0 r_i^n + a_1 r_i^{n+1} + a_2 r_i^{n+2} + ... + a_k r_i^{n+k} = r_i^n \left(a_0 + a_1 r_i + a_2 r_i^2 + ... + a_k r_i^k\right) = 0$$

Dacă rădăcinile $r_i$ $(i = 1, 2, ..., k)$ sunt distincte, atunci relaţia de recurenţă are soluţia generală

$$x_n = c_1 r_1^n + c_2 r_2^n + ... + c_k r_k^n \tag{6.1.4}$$

unde coeficienţii $c_1, c_2, ..., c_k$ se pot determina din condiţiile iniţiale.

• **Ecuaţia caracteristică admite rădăcini reale multiple**

Fie $r$ o rădăcină multiplă de ordinul $p$ a ecuaţiei caracteristice. Atunci

$$r^n, nr^n, n^2 r^n, ..., n^{p-1} r^n$$

sunt soluţii liniar independente ale relaţiei de recurenţă şi

$$x_n = \left(c_1 + c_2 n + ... + c_{p-1} n^{p-1}\right) r^n \tag{6.1.5}$$

este o soluţie a relaţiei de recurenţă. Acest lucru se mai poate demonstra uşor dacă ţinem cont de faptul că o rădăcină multiplă de ordinul $p$ a unui polinom $P(x)$ este rădăcină şi a polinoamelor derivate $P'(x), P''(x), ..., P^{(p-1)}(x)$.

Soluţia generală este suma dintre soluţia generală corespunzătoare rădăcinilor simple ale ecuaţiei caracteristice şi soluţia generală corespunzătoare rădăcinilor multiple.

Dacă ecuaţia caracteristică are rădăcinile simple $r_1, r_2, ..., r_s$ şi rădăcinile multiple $r_{s_1}, r_{s+2}, ..., r_{s+t}$ de multiplicitate $p_1, p_2, ..., p_t$ $(s+p_1+p_2+...+p_t = k)$, atunci soluţia generală a relaţiei de recurenţă este

$$\begin{aligned} x_n &= c_1 r_1^n + c_2 r_2^n + ... + c_s r_s^n + \\ &\quad \left(c_1^{(1)} + c_2^{(1)} n + ... + c_{p_1-1}^{(1)} n^{p_1-1}\right) + \\ &\quad ... \\ &\quad \left(c_1^{(t)} + c_2^{(t)} n + ... + c_{p_t-1}^{(1)} n^{p_t-1}\right) + \end{aligned}$$

unde $c_1, ..., c_s, c_1^{(1)}, ..., c_{p_1-1}^{(1)}, ..., c_1^{(t)}, ..., c_{p_t-1}^{(t)}$ sunt constante, care se pot determina din condiţiile iniţiale.

• **Ecuaţia caracteristică admite rădăcini complexe simple**

Fie $r = ae^{ib} = a(\cos b + i \sin b)$ o rădăcină complexă. Ecuaţia caracteristică are coeficienţi reali, deci şi conjugata $\bar{r} = ae^{-ib} = a(\cos b - i \sin b)$ este rădăcină pentru ecuaţia caracteristică. Atunci soluţiile corespunzătoare acestora în sistemul fundamental de soluţii pentru recurenţa liniară şi omogenă sunt

$$x_n^{(1)} = a^n \cos bn,\ x_n^{(2)} = a^n \sin bn.$$

• **Ecuaţia caracteristică admite rădăcini complexe multiple** Dacă ecuaţia caracteristică admite perechea de rădăcini complexe

$$r = ae^{ib}, \bar{r} = ae^{-ib} \quad b \neq 0$$

de ordin de multiplicitate $k$, atunci soluţiile corespunzătoare acestora în sistemul fundamental de soluţii sunt

$$x_n^{(1)} = a^n \cos bn,\ x_n^{(2)} = na^n \cos bn,\ ...,\ x_n^{(k)} = n^{k-1} a^n \cos bn,$$

$$x_n^{(k+1)} = a^n \sin bn,\ x_n^{(k+2)} = na^n \sin bn,\ ...,\ x_n^{(2k)} = n^{k-1} a^n \sin bn,$$

Pentru a obţine soluţia generală a recurenţei omogene de ordinul $n$ cu coeficienţi constanţi se procedează astfel:

1. Se determină rădăcinile ecuaţiei caracteristice
2. Se scrie contribuţia fiecărei rădăcini la soluţia generală.
3. Se însumează şi se obţine soluţia generală în funcţie de $n$ constante arbitrare.
4. Dacă sunt precizate condiţiile iniţiale atunci se determină constantele şi se obţine o soluţie unică.

## 6.2 Ecuaţii recurente neomogene

### 6.2.1 O formă simplă

Considerăm acum recurenţe de următoarea formă mai generală

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

unde $b$ este o constantă, iar $p(n)$ este un polinom în $n$ de grad $d$. Ideea generală este să reducem un astfel de caz la o formă omogenă.

De exemplu, o astfel de recurenţă poate fi:

$$t_n - 2t_{n-1} = 3^n$$

În acest caz, $b = 3$ şi $p(n) = 1$. Înmulţim recurenţa cu 3, şi obţinem

$$3t_n - 6t_{n-1} = 3^{n+1}$$

Înlocuind pe $n$ cu $n + 1$ în recurenţa iniţială, avem

$$t_{n+1} - 2t_n = 3^{n+1}$$

Scădem aceste două ecuaţii

$$t_{n+1} - 5t_n + 6t_{n-1} = 0$$

Am obţinut o recurenţă omogenă. Ecuaţia caracteristică este:

$$x^2 - 5x + 6 = 0$$

adică $(x - 2)(x - 3) = 0$. Intuitiv, observăm că factorul $(x - 2)$ corespunde părţii stângi a recurenţei iniţiale, în timp ce factorul $(x - 3)$ a apărut ca rezultat al calculelor efectuate pentru a scăpa de partea dreaptă.

Generalizând acest procedeu, se poate arăta că, pentru a rezolva ecuaţia iniţială, este suficient să luăm următoarea ecuaţie caracteristică:

$$(a_0 x^k + a_1 x^{k-1} + \ + a_k)(x - b)^{d+1} = 0$$

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

Vom rezolva acum recurenţa corespunzatoare problemei turnurilor din Hanoi:

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

iar $t_0 = 0$. Rescriem recurenţa astfel

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

care este de forma generală prezentată la început, cu $b = 1$ si $p(n) = 1$. Ecuaţia caracteristică este atunci $(x-2)(x-1) = 0$, cu soluţiile 1 şi 2. Soluţia generală a recurenţei este:

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

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

$$t_1 = 2t_0 + 1 = 1.$$

Din condiţiile iniţiale, obţinem

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

Dacă ne interesează doar ordinul lui $t_n$, nu este necesar să calculăm efectiv constantele în soluţia generală. Dacă ştim că $t_n = c_1 1^n + c_2 2^n$, rezultă $t_n \in O(2^n)$.

Din faptul că numărul de mutări a unor discuri nu poate fi negativ sau constant, deoarece avem în mod evident $t_n \geq n$, deducem că $c_2 > 0$. Avem atunci $t_n \in \Omega(2^n)$ şi deci, $t_n \in \Theta(2^n)$. Putem obţine chiar ceva mai mult. Substituind soluţia generală înapoi în recurenţa iniţială, găsim

$$1 = t_n - 2t_{n-1} = c_1 + c_2 2^n - 2(c_1 + c_2 2^{n-1}) = -c1$$

Indiferent de condiţia iniţială, $c_1$ este deci $-1$.

### 6.2.2 O formă mai generală

O ecuaţie recurentă neomogenă de formă mai generală este:

$$\sum_{j=0}^{k} a_j Tn - j = b_1^n \cdot p_{d_1}(n) + b_2^n \cdot p_{d_2}(n) + ...$$

în care

$$p_d(n) = n^d + c_1 n^{d-1} + ... + c_d$$

Ecuaţia caracteristică completă este:

$$\left( \sum_{j=0}^{k} a_j \cdot r^{k-j} \right) \cdot (r - b_1)^{d_1+1} \cdot (r - b_2)^{d_2+1} \cdot ... = 0$$

Exemplul 3: $T_n = 2T(n-1) + n + 2^n, n \geq 1, T_0 = 0$.

Acestui caz îi corespund $b_1 = 1$, $p_1(n) = n$, $d_1 = 1$ şi $b_2 = 2$, $p_2(n) = 1$, $d_2 = 0$, iar ecuaţia caracteristică completă este:

$$(r-2)(r-1)^2(r-2) = 0$$

cu soluţia:

$$T(n) = c_1 \cdot 1^n + c_2 \cdot n \cdot 2^n + c_3 \cdot 2n + c_4 \cdot n \cdot 2^n$$

$$\begin{cases} T(0) = 0 \\ T(1) = 2T(0) + 1 + 2^1 = 3 \\ T(2) = 2T(1) + 2 + 2^2 = 12 \\ T(3) = 2T(2) + 3 + 2^3 = 35 \end{cases} \quad \begin{cases} c_1 + c_3 = 0 \\ c_1 + c_2 + 2c_3 + 2c_4 = 3 \\ c_1 + 2c_2 + 4c_3 + 8c_4 = 12 \\ c_1 + 3c_2 + 8c_3 + 24c_4 = 35 \end{cases} \Rightarrow \begin{cases} c_1 = -2 \\ c_2 = -1 \\ c_3 = 2 \\ c_4 = 1 \end{cases}$$

Deci

$$T(n) = -2 - n + 2^{n+1} + n \cdot 2^n = O(n \cdot 2^n).$$

### 6.2.3 Teorema master

De multe ori apare relaţia de recurenţă de forma

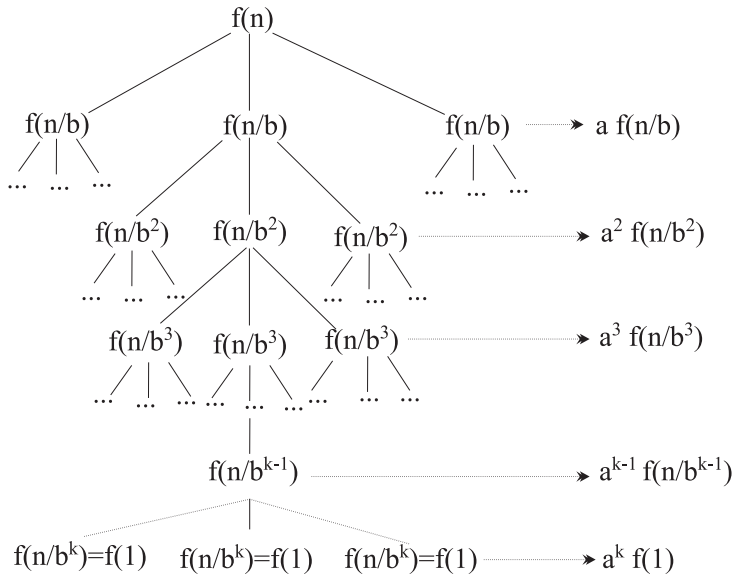
$$T(n) = aT(n/b) + f(n) \tag{6.2.6}$$

unde $a$ şi $b$ sunt constante iar $f(n)$ este o funcţie (aplicarea metodei *Divide et Impera* conduce de obicei la o astfel de ecuaţie recurentă). Aşa numita teoremă Master dă o metodă generală pentru rezolvarea unor astfel de recurenţe când $f(n)$ este un simplu polinom. Soluţia dată de teorema master este:

1. dacă $f(n) = O\left(n^{\log_b(a-\varepsilon)}\right)$ cu $\varepsilon > 0$ atunci $T(n) = \Theta\left(n^{\log_b a}\right)$
2. dacă $f(n) = \Theta\left(n^{\log_b a}\right)$ atunci $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$
3. dacă $f(n) = \Omega\left(n^{\log_b(a+\varepsilon)}\right)$ şi $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ cu $c < 1$ atunci $T(n) = \Theta\left(f(n)\right)$.

Din păcate, teorema Master nu funcţionează pentru toate funcţiile $f(n)$, şi multe recurenţe utile nu sunt de forma (6.2.6). Din fericire însă, aceasta este o tehnică de rezolvare a celor mai multe relaţii de recurenţă provenite din metoda *Divide et Impera*.

Pentru a rezolva astfel de ecuaţii recurente vom reprezenta arborele generat de ecuaţia recursivă. Rădăcina arborelui conţine valoarea $f(n)$, şi ea are noduri descendente care sunt noduri rădăcină pentru arborele provenit din $T(n/b)$.




Pe nivelul $i$ se află nodurile care conţin valoarea $a^i f(n/b^i)$. Recursivitatea se opreşte când se obţine un *caz de bază* pentru recurenţă.

Presupunem că $T(1) = f(1)$.

Cu această reprezentare este foarte clar că $T(n)$ este suma valorilor din nodurile arborelui. Presupunând că fiecare nivel este plin, obţinem

$$T(n) = f(n) + af(n/b) + a^2 f(n/b^2) + a^3 f(n/b^3) + ... + a^k f(n/b^k)$$

unde $k$ este adâncimea arborelui de recursivitate. Din $n/b^k = 1$ rezultă $k = \log_b n$. Ultimul termen diferit de zero în sumă este de forma $a^k = a^{\log_b n} = n^{\log_b a}$ (ultima egalitate fiind întâlnită în liceu!).

Acum putem uşor enunţa şi demonstra teorema Master.

**Teorema 1** *(***Teorema Master***) Relaţia de recurenţă* $T(n) = aT(n/b) + f(n)$ *are următoarea soluţie:*

- *dacă* $af(n/b) = \alpha f(n)$ *unde* $\alpha < 1$ *atunci* $T(n) = \Theta(f(n))$*;*
- *dacă* $af(n/b) = \beta f(n)$ *unde* $\beta > 1$ *atunci* $T(n) = \Theta(n^{\log_b a})$*;*
- *dacă* $af(n/b) = f(n)$ *atunci* $T(n) = \Theta(f(n) \log_b n)$*;*

*Demonstraţie*: Dacă $f(n)$ este un factor constant mai mare decât $f(b/n)$, atunci prin inducţie se poate arăta că suma este a unei progresii geometrice descrescătoare. Suma în acest caz este o constantă înmulţită cu primul termen care este $f(n)$.

Dacă $f(n)$ este un factor constant mai mic decât $f(b/n)$, atunci prin inducţie se poate arăta că suma este a unei progresii geometrice crescătoare. Suma în acest caz este o constantă înmulţită cu ultimul termen care este $n^{\log_b a}$.

Dacă $af(b/n) = f(n)$, atunci prin inducţie se poate arăta că fiecare din cei $k+1$ termeni din sumă sunt egali cu $f(n)$.

**Exemple**.

1. Selecţia aleatoare: $T(n) = T(3n/4) + n$.
Aici $af(n/b) = 3n/4$ iar $f(n) = n$, rezultă $\alpha = 3/4$, deci $T(n) = \Theta(n)$.

2. Algoritmul de multiplicare al lui Karatsuba: $T(n) = 3T(n/2) + n$.
Aici $af(n/b) = 3n/2$ iar $f(n) = n$, rezultă $\alpha = 3/2$, deci $T(n) = \Theta(n^{\log_2 3})$.

3. Mergesort: $T(n) = 2T(n/2) + n$.
Aici $af(n/b) = n$, iar $f(n) = n$, rezultă $\alpha = 1$ deci $T(n) = \Theta(n \log_2 n)$.

Folosind aceaşi tehnică a arborelui recursiv, putem rezolva recurenţe pentru care nu se poate aplica teorema Master.

### 6.2.4 Transformarea recurenţelor

La *Mergesort* am avut o relaţe de recurenţă de forma $T(n) = 2T(n/2) + n$ şi am obţinut soluţia $T(n) = O(n \log_2 n)$ folosind teorema Master (metoda arborelui de recursivitate). Această modalitate este corectă dacă $n$ este o putere a lui 2, dar pentru alte valori ale lui $n$ această recurenţă nu este corectă. Când $n$ este impar, recurenţa ne cere să sortăm un număr elemente care nu este întreg! Mai rău chiar, dacă $n$ nu este o putere a lui 2, nu vom atinge niciodată cazul de bază $T(1) = 0$.

Pentru a obţine o recurenţă care să fie validă pentru orice valori întregi ale lui $n$, trebuie să determinăm cu atenţie marginile inferioară şi superioară:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

Metoda *transformării domeniului* rescrie funcţia $T(n)$ sub forma $S(f(n))$, unde $f(n)$ este o funcţie simplă şi $S()$ are o recurenţă mai uşoară.

Următoarele inegalităţi sunt evidente:

$$T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n.$$

Acum definim o nouă funcţie $S(n) = T(n + \alpha)$, unde $\alpha$ este o constantă necunoscută, aleasă astfel încât să fie satisfăcută recurenţa din teorema Master $S(n) \leq S(n/2) + O(n)$. Pentru a obţine valoarea corectă a lui $\alpha$, vom compara două versiuni ale recurenţei pentru funcţia $S(n + \alpha)$:

$$\begin{cases} S(n) \leq 2S(n/2) + O(n) & \Rightarrow T(n+\alpha) \leq 2T(n/2+\alpha) + O(n) \\ T(n) \leq 2T(n/2+1) + n & \Rightarrow T(n+\alpha) \leq 2T((n+\alpha)/2+1) + n + \alpha \end{cases}$$

Pentru ca aceste două recurenţe să fie egale, trebuie ca $n/2+\alpha = (n+\alpha)/2+1$, care implică $\alpha = 2$. Teorema Master ne spune acum că $S(n) = O(n \log n)$, deci

$$T(n) = S(n-2) = O((n-2)\log(n-2) = O(n \log n).$$

Un argument similar dă o ajustare a marginii inferioare $T(n) = \Omega(n \log n)$.

Deci, $T(n) = \Theta(n \log n)$ este un rezultat întemeiat deşi am ignorat marginile inferioară şi superioară de la început!

Transformarea domeniului este utilă pentru înlăturarea marginilor inferioară şi superioară, şi a termenilor de ordin mic din argumentele oricărei recurenţe care se potriveşte un pic cu teorema master sau metoda arborelui de recursivitate.

Există în geometria computaţională o structură de date numită *arbore pliat*, pentru care costul operatiei de căutare îndeplineşte relaţia de recurenţă

$$T(n) = T(n/2) + T(n/4) + 1.$$

Aceasta nu se potriveşte cu teorema master, pentru că cele două subprobleme au dimensiuni diferite, şi utilizând metoda arborelui de recursivitate nu obţinem decât nişte margini slabe $\sqrt{n} << T(n) << n$.

Dacă nu au forma standard, ecuaţiile recurente pot fi aduse la această formă printr-o schimbare de variabilă. O schimbare de variabilă aplicabilă pentru ecuaţii de recurenţă de tip multiplicativ este:

$$n = 2^k \Leftrightarrow k = \log n$$

De exemplu, fie

$$T(n) = 2 \cdot T(n/2) + n \cdot \log n, n > 1$$

Facem schimbarea de variabilă $t(k) = T(2^k)$ şi obţinem:

$$t(k) - 2 \cdot t(k-1) = k \cdot 2^k, \text{ deci } b = 2, p(k) = k, d = 1$$

Ecuaţia caracteristică completă este:

$$(r-2)^3 = 0$$

cu soluţia

$$t(k) = c_1 \cdot 2^k + c_2 \cdot k \cdot 2^k + c_3 \cdot k^2 \cdot 2^k$$

Deci

$$T(n) = c_1 \cdot n + c_2 \cdot n \cdot \log n + c_3 \cdot n \cdot \log^2 n \in O(n \cdot \log^2 n | n = 2^k)$$

Uneori, printr-o schimbare de variabilă, putem rezolva recurenţe mult mai complicate. În exemplele care urmează, vom nota cu $T(n)$ termenul general al recurenţei si cu $t_k$ termenul noii recurenţe obţinute printr-o schimbare de variabilă.

Presupunem pentru început că $n$ este o putere a lui 2.

Un prim exemplu este recurenta

$$T(n) = 4T(n/2) + n, \quad n > 1$$

în care înlocuim pe $n$ cu $2^k$, notam $t_k = T(2^k) = T(n)$ şi obţinem

$$t_k = 4t_{k-1} + 2^k$$

Ecuaţia caracteristică a acestei recurenţe liniare este

$$(x-4)(x-2) = 0$$

şi deci, $t_k = c_1 4^k + c_2 2^k$. Înlocuim la loc pe $k$ cu $\log_2 n$

$$T(n) = c_1 n^2 + c_2 n$$

Rezultă

$$T(n) \in O(n^2 | n \text{ este o putere a lui } 2)$$

Un al doilea exemplu îl reprezintă ecuaţia

$$T(n) = 4T(n/2) + n^2, \quad n > 1$$

Procedând la fel, ajungem la recurenta

$$t_k = 4t_{k-1} + 4^k$$

cu ecuaţia caracteristică

$$(x-4)^2 = 0$$

şi soluţia generală $t_k = c_1 4^2 + c_2 k 4^2$.

Atunci,

$$T(n) = c_1 n^2 + c_2 n^2 \lg n$$

şi obţinem

$$T(n) \in O(n^2 \log n | n \text{ este o putere a lui } 2)$$

În sfârşit, să considerăm şi exemplul

$$T(n) = 3T(n/2) + cn, \quad n > 1$$

$c$ fiind o constantă. Obţinem succesiv

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$t_k = 3t_{k-1} + c2^k$$

cu ecuaţia caracteristică

$$(x-3)(x-2) = 0$$
$$t_k = c_1 3^k + c_2 2^k$$
$$T(n) = c_1 3^{\lg n} + c_2 n$$

şi, deoarece

$$a^{\lg b} = b^{\lg a}$$

obţinem

$$T(n) = c_1 n^{\lg 3} + c_2 n$$

deci,

$$T(n) \in O(n^{\lg 3} | n \text{ este o putere a lui } 2)$$

Putem enunţa acum o proprietate care este utilă ca reţetă pentru analiza algoritmilor cu recursivităţi de forma celor din exemplele precedente.

Fie $T : \mathbb{N} \longrightarrow \mathbb{R}^+$ o funcţie eventual nedescrescătoare

$$T(n) = aT(n/b) + cn^k, \quad n > n_0$$

unde: $n_0 \geq 1$, $b \geq 2$ si $k \geq 0$ sunt întregi; $a$ şi $c$ sunt numere reale pozitive; $n/n_0$ este o putere a lui $b$. Atunci avem

$$T(n) \in \begin{cases} \Theta(n^k), & \text{pentru } a < b^k; \\ \Theta(n^k \log n), & \text{pentru } a = b^k; \\ \Theta(n^{\log_b a}), & \text{pentru } a > b^k; \end{cases}$$

## 6.3 Probleme rezolvate

1. Să se rezolve ecuaţia:

$$F_{n+2} = F_{n+1} + F_n, \; F_0 = 0, \; F_1 = 1.$$

Ecuaţaia caracteristică corespunzătoare

$$r^2 - r - 1 = 0$$

are soluţiile

$$r_1 = \frac{1+\sqrt{5}}{2}, \; r_2 = \frac{1-\sqrt{5}}{2}.$$

Soluţia generală este

$$F_n = c_1 \left( \frac{1+\sqrt{5}}{2} \right)^n + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^n.$$

Determinăm constantele $c_1$ şi $c_2$ din condiţiile iniţiale $F_0 = 0$ şi $F_1 = 1$. Rezolvând sistemul

$$\begin{cases} c_1 + c_2 = 0 \\ c_1 \left( \frac{1+\sqrt{5}}{2} \right) + c_2 \left( \frac{1-\sqrt{5}}{2} \right) = 1 \end{cases}$$

obţinem $c_1 = \frac{1}{\sqrt{5}}$ şi $c_1 = -\frac{1}{\sqrt{5}}$.

Deci, soluţia relaţiei de recurenţă care defineşte numerele lui Fibonacci este:

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

2. Să se rezolve relaţia de recurenţă:

$$x_{n+3} = x_{n+2} + 8x_{n+1} - 12x_n, \quad x_0 = 0, x_1 = 2, x_2 = 3.$$

Ecuaţia caracteristică corespunzătoare este:

$$r^3 - r^2 - 8r + 12 = 0$$

şi are soluţiile: $r_1 = r_2 = 2$ şi $r_3 = -3$.

Soluţia generală este de forma:

$$x_n = (c_1 + nc_2)2^n + c_3(-3)^n.$$

Din condiţiile iniţiale rezultă constantele: $c_1 = \frac{1}{5}$, $c_2 = \frac{1}{2}$ şi $c_3 = -\frac{1}{5}$.

Soluţia generală este:

$$x_n = \left(\frac{n}{2} + \frac{1}{5}\right)2^n - \frac{1}{5}(-3)^n.$$

3. Să se rezolve relaţia de recurenţă:

$$x_{n+3} = 6x_{n+2} - 12x_{n+1} + 8x_n, \quad x_0 = 0, x_1 = 2, x_2 = 4.$$

Ecuaţia caracteristică corespunzătoare este:

$$r^3 - 6r^2 + 12r - 8 = 0$$

şi are soluţiile: $r_1 = r_2 = r_3 = 2$.

Soluţia generală este de forma:

$$x_n = (c_1 + c_2 n + c_3 n^2)2^n.$$

Din condiţiile iniţiale rezultă constantele: $c_1 = 0$, $c_2 = \frac{3}{2}$ şi $c_3 = -\frac{1}{2}$.

Soluţia generală este:

$$x_n = \left(\frac{3}{2}n - \frac{1}{2}n^2\right)2^n = (3n - n^2)2^{n-1}.$$

4. Să se rezolve relaţia de recurenţă:

$$x_{n+2} = 2x_{n+1} - 2x_n, \quad x_0 = 0, x_1 = 1.$$

Ecuaţia caracteristică corespunzătoare este:

$$r^2 - 2r + 2 = 0$$

şi are soluţiile: $r_1 = 1 + i$ şi $r_2 = 1 - i$ care se pot scrie sub formă trigonometrică astfel:

$$r_1 = \sqrt{2}\left(\cos\frac{\pi}{4} + i\sin\frac{\pi}{4}\right), r_2 = \sqrt{2}\left(\cos\frac{\pi}{4} - i\sin\frac{\pi}{4}\right).$$

Soluţiile fundamentale sunt:

$$x_n^{(1)} = \left(\sqrt{2}\right)^n \cos\frac{n\pi}{4}, x_n^{(2)} = \left(\sqrt{2}\right)^n \sin\frac{n\pi}{4}.$$

Soluţia generală este de forma:

$$x_n = \left(\sqrt{2}\right)^n \left(c_1\cos\frac{n\pi}{4} + c_2\sin\frac{n\pi}{4}\right).$$

Din condiţiile iniţiale rezultă constantele: $c_1 = 0$ si $c_2 = 1$.
Soluţia generală este:

$$x_n = \left(\sqrt{2}\right)^n \sin\frac{n\pi}{4}.$$

5. Să se rezolve relaţia de recurenţă:

$$x_{n+3} = 4x_{n+2} - 6x_{n+1} + 4x_n, \quad x_0 = 0, x_1 = 1, x_2 = 1.$$

Ecuaţia caracteristică corespunzătoare este:

$$r^3 - 4r^2 + 6r - 4 = 0$$

şi are soluţiile: $r_1 = 2$, $r_2 = 1 + i$ şi $r_3 = 1 - i$.
Soluţia generală este de forma:

$$x_n = c_1 2^n + c_2\left(\sqrt{2}\right)^n \cos\frac{n\pi}{4} + c_3\left(\sqrt{2}\right)^n \sin\frac{n\pi}{4}.$$

Din condiţiile iniţiale rezultă constantele: $c_1 = -\frac{1}{2}$, $c_2 = \frac{1}{2}$ si $c_3 = \frac{3}{2}$.
Soluţia generală este:

$$x_n = -2^{n-1} + \frac{\left(\sqrt{2}\right)^n}{2}\left(\cos\frac{n\pi}{4} + 3\sin\frac{n\pi}{4}\right).$$

6. Să se rezolve relaţia de recurenţă:

$$T(n) - 3T(n-1) + 4T(n-2) = 0, n \geq 2, T(0) = 0, T(1) = 1.$$

Ecuaţia caracteristică $r^2 - 3r + 4 = 0$ are soluţiile $r_1 = -1$, $r_2 = 4$, deci

$$T(n) = c_1(-1)^n + c_2 4^n$$

Constantele se determină din condiţiile iniţiale:

$$\begin{cases} c_1 + c_2 = 0 \\ -c_1 + 4c_2 = 1 \end{cases} \Rightarrow \begin{cases} c_1 = -\frac{1}{5} \\ c_2 = \frac{1}{5} \end{cases}$$

Soluţia este:

$$T(n) = \frac{1}{5}\left[4^n - (-1)^n\right].$$

7. Să se rezolve relaţia de recurenţă:

$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3), n \geq 3,$ cu $T(0) = 0, T(1) = 1, T(2) = 2.$

Ecuaţia caracteristică:

$$r^3 - 5r^2 + 8r - 4 = 0 \Rightarrow r_1 = 1, r_2 = r_3 = 2$$

deci

$$T(n) = c_1 1^n + c_2 2^n + c_3 n 2_n$$

Determinarea constantelor

$$\begin{cases} c_1 + c_2 & = 0 \\ c_1 + 2c_2 + 2c_3 & = 1 \\ c_1 + 4c_2 + 8c_3 & = 2 \end{cases} \Rightarrow \begin{cases} c_1 = -2 \\ c_2 = 2 \\ c_3 = -\frac{1}{2} \end{cases}$$

Deci

$$T(n) = -2 + 2^{n+1} - \frac{n}{2}2^n = 2^{n+1} - n2^{n-1} - 2.$$

8. Să se rezolve relaţia de recurenţă:

$$T(n) = 4T(n/2) + n \lg n.$$

În acest caz, avem $af(n/b) = 2n \lg n - 2n$, care nu este tocmai dublul lui $f(n) = n \lg n$. Pentru $n$ suficient de mare, avem $2f(n) > af(n/b) > 1.9f(n)$.

Suma este mărginită şi inferior şi superior de către serii geometrice crescătoare, deci soluţia este $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$. Acest truc nu merge în cazurile doi şi trei ale teoremei Master.

9. Să se rezolve relaţia de recurenţă:

$$T(n) = 2T(n/2) + n \lg n.$$

Nu putem aplica teorema Master pentru că $af(n/b) = n/(\lg n - 1)$ nu este egală cu $f(n) = n/\lg n$, iar diferenţa nu este un factor constant.

Trebuie să calculăm suma pe fiecare nivel şi suma totală în alt mod. Suma tuturor nodurilor de pe nivelul $i$ este $n/(\lg n - i)$. În particular, aceasta înseamnă că adâncimea arborelui este cel mult $\lg n - 1$.

$$T(n) = \sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} = \sum_{j=1}^{\lg n} \frac{n}{j} = nH_{\lg n} = \Theta(n \lg \lg n).$$

10. (Quicksort aleator). Să se rezolve relaţia de recurenţă:

$$T(n) = T(3n/4) + T(n/4) + n.$$

În acest caz nodurile de pe acelaşi nivel al arborelui de recursivitate au diferite valori. Nodurile din orice nivel *complet* (adică, deasupra oricărei frunze) au suma $n$, deci este la fel ca în ultimul caz al teoremei Master şi orice frunză are nivelul între $\log_4 n$ şi $\log_{4/3} n$.

Pentru a obţine o margine superioară, vom supraevalua $T(n)$ ignorând cazurile de bază şi extinzând arborele în jos către nivelul celei mai adânci frunze.

Similar, pentru a obţine o margine inferioară pentru $T(n)$, vom subevalua $T(n)$ contorizând numai nodurile din arbore până la nivelul frunzei care este cea mai puţin adâncă. Aceste observaţii ne dau marginile inferioară şi superioară:

$$n \log_4 n \leq T(n) \leq n \log_{4/3} n.$$

Deoarece aceste margini diferă numai printr-un factor constant, avem că $T(n) = \Theta(n \log n)$.

11. (Selecţie deterministă). Să se rezolve relaţia de recurenţă:

$$T(n) = T(n/5) + T(7n/10) + n.$$

Din nou, avem un arbore recursiv "trunchiat". Dacă ne uităm numai la nivelurile complete ale arborelui, observăm că suma pe nivel formează o serie geometrică descrescătoare $T(n) = n + 9n/10 + 81n/100 + ...$, deci este ca în primul caz al teoremei Master. Putem să obţinem o margine superioară ignorând cazurile de bază în totalitate şi crescând arborele spre infinit, şi putem obţine o margine inferioară contorizând numai nodurile din nivelurile complete. În ambele situaţii, seriile geometrice sunt majorate de termenul cel mai mare, deci $T(n) = \Theta(n)$.

12. Să se rezolve relaţia de recurenţă:

$$T(n) = 2\sqrt{n} \cdot T(\sqrt{n}) + n.$$

Avem cel mult $\lg \lg n$ niveluri dar acum avem nodurile de pe nivelul $i$ care au suma $2^i n$. Avem o serie geometrică crescătoare a sumelor nivelurilor, la fel ca

în cazul doi din teorema Master, deci $T(n)$ este majorată de suma nivelurilor cele mai adânci. Se obţine:

$$T(n) = \Theta(2^{\lg \lg n} n) = \Theta(n \log n).$$

13. Să se rezolve relaţia de recurenţă:

$$T(n) = 4\sqrt{n} \cdot T(\sqrt{n}) + n.$$

Suma nodurilor de pe nivelul $i$ este $4^i n$. Avem o serie geometrică crescătoare, la fel ca în cazul doi din teorema master, deci nu trebuie decât să avem grijă de aceste niveluri. Se obţine

$$T(n) = \Theta(4^{\lg \lg n} n) = \Theta(n \log^2 n).$$

# Capitolul 7

# Algoritmi elementari

## 7.1 Operaţii cu numere

### 7.1.1 Minim şi maxim

Să presupunem că dorim să determinăm valorile minimă şi maximă dintru-un vector $x[1..n$. Procedăm astfel:

```
vmin = x[1];
vmax = x[1];
for i=2, n
  vmin = minim(vmin, x[i])
  vmax = maxim(vmax, x[i])
```

Evident se fac $2n-2$ comparaţii. Se poate mai repede? Da! Împărţim şirul în două şi determinăm $vmin$ şi $vmax$ în cele două zone. Comparăm $vmin1$ cu $vmin2$ şi stabilim $vminm$. La fel pentru $vmax$. Prelucrarea se repetă pentru cele două zone (deci se foloseşte recursivitatea). Apar câte două comparaţii în plus de fiecare dată. Dar câte sunt în minus? Presupunem că $n$ este o putere a lui 2 şi $T(n)$ este numărul de comparaţii. Atunci

$$T(n) = 2T(n/2) + 2 \text{ şi } T(2) = 1.$$

Cum rezolvăm această relaţie de recurenţă? Bănuim că soluţia este de forma $T(n) = an + b$. Atunci $a$ şi $b$ trebuie să satisfacă sistemul de ecuaţii

$$\begin{cases} 2a + b = 1 \\ an + b = 2(an/2 + b) + 2 \end{cases}$$

care are soluţia $b = -2$ şi $a = 3/2$, deci (pentru $n$ putere a lui 2), $T(n) = 3n/2 - 2$, adică 75% din algoritmul anterior. Se poate demonstra că numărul de comparaţii este $3\lceil n/2 \rceil - 2$ pentru a afla minimum şi maximum.

O idee similară poate fi aplicată pentru varianta secvenţială. Presupunem că şirul are un număr par de termeni. Atunci, algoritmul are forma:

```
vmin = minim(x[1],x[2])
vmax = maxim(x[1],x[2])
for(i=3;i<n;i=i+2)
  cmin = minim(x[i],x[i+1])
  cmax = maxim(x[i],x[i+1])
  if cmin < vmin
    vmin = cmin
  if vmax > cmax
    vmax = cmax
```

Fiecare iteraţie necesită trei comparaţii, iar iniţializarea variabilelor necesită o comparaţie. Ciclul se repetă de $(n-2)/2$ ori, deci avem un total de $3n/2 - 2$ comparaţii pentru $n$ par.

### 7.1.2 Divizori

Fie $n$ un număr natural. Descompunerea în facori primi

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot ... \cdot p_k^{\alpha_k} \tag{7.1.1}$$

se numeşte *descompunere canonică*. Dacă notăm prin $d(n)$ numărul divizorilor lui $n \in \mathbb{N}$, atunci:

$$d(n) = (1 + \alpha_1)(1 + \alpha_2)...(1 + \alpha_k) \tag{7.1.2}$$

Pentru calculul lui $d(n)$ putem folosi următorul algoritm:

```
static int ndiv(int n)
{
  int d,p=1,nd;

  d=2;nd=0;
  while(n%d==0){nd++;n=n/d;}
  p=p*(1+nd);

  d=3;
  while(d*d<=n)
  {
    nd=0;
    while(n%d==0){nd++;n=n/d;}
```

```
    p=p*(1+nd);
    d=d+2;
  }
  if(n!=1) p=p*2;
  return p;
}
```

### 7.1.3 Numere prime

Pentru testarea primalităţi unui număr putem folosi următorul algoritm:

```
static boolean estePrim(int nr)
{
  int d;
  if(nr<=1) return false;
  if(nr==2) return true;
  if(nr%2==0) return false;
  d=3;
  while((d*d<=nr)&&(nr%d!=0)) d=d+2;
  if(d*d>nr) return true; else return false;
}
```

## 7.2 Algoritmul lui Euclid

### 7.2.1 Algoritmul clasic

Un algoritm pentru calculul celui mai mare divizor comun (*cmmdc*) a două numere naturale poate fi descompunerea lor în factori şi calculul produsului tuturor divizorilor comuni. De exemplu dacă $a = 1134 = 2 * 3 * 3 * 3 * 3 * 7$ şi $b = 308 = 2 * 2 * 7 * 11$ atunci $cmmdc(a, b) = 2 * 7 = 14$.

Descompunerea în factori a unui număr natural $n$ poate necesita încercarea tuturor numerelor naturale din intervalul $[2, \sqrt{n}]$.

Un algoritm eficient pentru calculul $cmmdc(a, b)$ este algoritmul lui Euclid.

```
static int cmmdc(int a, int b}
{
  int c;
  if (a < b) { c = a; a = b; b = c; }
  while((c=a%b) != 0) { a = b; b = c;}
  return b;
}
```

Pentru $a = 1134$ şi $b = 308$ se obţine:

$a_0 = 1134, \quad b_0 = 308;$
$a_1 = 308, \quad b_1 = 210;$
$a_2 = 210, \quad b_2 = 98;$
$a_3 = 98, \quad b_3 = 14.$

**Lema 1** $cmmdc(a - x * b, b) = cmmdc(a, b)$.

*Demonstraţie*: Pentru început arătăm că $cmmdc(a - x * b, b) >= cmmdc(a, b)$. Presupunem că $d$ divide $a$ ş $b$, deci $a = c_1 * d$ şi $b = c_2 * d$. Atunci $d$ divide $a - x * b$ pentru că $a - x * b = (c_1 - x * c_2) * d$.

Demonstrăm şi inegalitatea contrară $cmmdc(a - x * b, b) <= cmmdc(a, b)$. Presupunem că $d$ divide $a - x * b$ şi $b$, deci $a - x * b = c_3 * d$ şi $b = c_2 * d$. Atunci $d$ divide $a$ pentru că $a = (a - x * b) + x * b = (c_3 + x * c_2) * d$. De aici rezultă că

$cmmdc(b, c) = cmmdc(c, b) = cmmdc(a \mod b, b) = gcd(a, b)$.

Prin inducţie rezultă că cel mai mare divizor comun al ultimelor două numere este egal cu cel mai mare divizor comun al primelor două numere. Dar pentru cele două numere $a$ şi $b$ din final, $cmmdc(a, b) = b$, pentru că $b$ divide $a$.

### 7.2.2 Algoritmul lui Euclid extins

Pentru orice două numere intregi pozitive, există $x$ şi $y$ (unul negativ) astfel încât $x * a + y * b = cmmdc(a, b)$. Aceste numere pot fi calculate parcurgând înapoi algoritmul clasic al lui Euclid.

Fie $a_k$ şi $b_k$ valorile lui $a$ şi $b$ după $k$ iteraţii ale buclei din algoritm. Fie $x_k$ şi $y_k$ numerele care indeplinesc relaţia $x_k * a_k + y_k * b_k = cmmdc(a_k, b_k) = cmmdc(a, b)$. Prin inductie presupunem că $x_k$ ş $y_k$ există, pentru că la sfârşit, când $b_k$ divide $a_k$, putem lua $x_k = 0$ ş $y_k = 1$.

Presupunând că $x_k$ şi $y_k$ sunt cunoscute, putem calcula $x_{k-1}$ şi $y_{k-1}$.

$a_k = b_{k-1}$ şi $b_k = a_{k-1} \mod b_{k-1} = a_{k-1} - d_{k-1} * b_{k-1}$, unde

$d_{k-1} = a_{k-1}/b_{k-1}$ (împărţire întreagă).

Substituind aceste expresii pentru $a_k$ şi $b_k$ obţinem

$$\begin{aligned} cmmdc(a, b) &= x_k * a_k + y_k * b_k \\ &= x_k * b_{k-1} + y_k * (a_{k-1} - d_{k-1} * b_{k-1}) \\ &= y_k * a_{k-1} + (x_k - y_k * d_{k-1}) * b_{k-1}. \end{aligned}$$

Astfel, ţinând cont de relaţia $x_{k-1} * a_{k-1} + y_{k-1} * b_{k-1} = cmmdc(a, b)$, obţinem

$x_{k-1} = y_k,$
$y_{k-1} = x_k - y_k * d_{k-1}.$

Pentru 1134 şi 308, obţinem:

$a_0 = 1134, \quad b_0 = 308, \quad d_0 = 3;$
$a_1 = 308, \quad b_1 = 210, \quad d_1 = 1;$
$a_2 = 210, \quad b_2 = 98, \quad d_2 = 2;$
$a_3 = 98, \quad b_3 = 14, \quad d_3 = 7.$

şi de asemenea, valorile pentru $x_k$ şi $y_k$:

$$\begin{array}{llllll} x_3 = 0, & y_3 & & = & 1; \\ x_2 = 1, & y_2 = & 0-1*2 & = & -2; \\ x_1 = -2, & y_1 = & 1+2*1 & = & 3; \\ x_0 = 3, & y_1 = & -2-3*3 & = & -11. \end{array}$$

Desigur relaţia $3*1134-11*308=14$ este corectă. Soluţia nu este unică. Să observăm că $(3+k*308)*1134-(11+k*1134)*308=14$, pentru orice $k$, ceea ce arată că valorile calculate pentru $x=x_0$ şi $y=y_0$ nu sunt unice.

## 7.3 Operaţii cu polinoame

Toate operaţiile cu polinoame obişnuite se fac utilizând şiruri de numere care reprezintă coeficienţii polinomului. Notăm cu $a$ si $b$ vectorii coeficienţilor polinoamelor cu care se operează şi cu $m$ şi $n$ gradele lor. Deci

$$a(X)=a_mX^m+...+a_1X+a_0 \text{ şi } b(X)=b_nX^n+...+b_1X+b_0.$$

### 7.3.1 Adunarea a două polinoame

Este asemănătoare cu adunarea numerelor mari.

```
static int[] sumap(int[] a, int[] b)
{
  int m,n,k,i,j,minmn;
  int[] s;
  m=a.length-1;
  n=b.length-1;
  if(m<n) {k=n; minmn=m;} else {k=m; minmn=n;}
  s=new int[k+1];
  for(i=0;i<=minmn;i++) s[i]=a[i]+b[i];
  if(minmn<m) for(i=minmn+1;i<=k;i++) s[i]=a[i];
         else for(i=minmn+1;i<=k;i++) s[i]=b[i];
  i=k;
  while((s[i]==0)&&(i>=1)) i--;
  if(i==k) return s;
    else
    {
      int[] ss=new int[i+1];
      for(j=0;j<=i;j++) ss[j]=s[j];
      return ss;
    }
}
```

### 7.3.2 Înmulţirea a două polinoame

Evident, gradul polinomului produs $p = a \cdot b$ este $m+n$ iar coeficientul $p_k$ este suma tuturor produselor de forma $a_i \cdot b_j$ unde $i + j = k$, $0 \leq i \leq m$ şi $0 \leq j \leq n$.

```
static int[] prodp(int[] a, int[] b)
{
  int m,n,i,j;
  int[] p;
  m=a.length-1;
  n=b.length-1;
  p=new int[m+n+1];
  for(i=0;i<=m;i++)
    for(j=0;j<=n;j++)
      p[i+j]+=a[i]*b[j];
  return p;
}
```

### 7.3.3 Calculul valorii unui polinom

Valoarea unui polinom se calculează eficient cu schema lui Horner:

$$a(x) = (...((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + ... + a_1) \cdot x + a_0$$

```
static int valp(int[] a, int x)
{
  int m,i,val;
  m=a.length-1;
  val=a[m];
  for(i=m-1;i>=0;i--)
    val=val*x+a[i];
  return val;
}
```

### 7.3.4 Calculul derivatelor unui polinom

Fie

$$b(X) = b_n X^n + b_{n-1} X^{n-1} + ... + b_1 X + b_0$$

derivata de ordinul 1 a polinomului

$$a(X) = a_m X^m + a_{m-1} X^{m-1} + ... + a_1 X + a_0.$$

Dar

$$a'(X) = m \cdot a_m \cdot X^{m-1} + (m-1) \cdot a_{m-1} \cdot X^{m-2} + ... + 2 \cdot a_2 \cdot X + a_1.$$

Rezultă că

$$n = m - 1$$

şi

$$b_i = (i+1) \cdot a_{i+1} \text{ pentru } 0 \leq i \leq n.$$

```
static int[] derivp(int[] a)
{
  int m,n,i;
  int[] b;
  m=a.length-1;
  n=m-1;
  b=new int[n+1];
  for(i=0;i<=n;i++)
    b[i]=(i+1)*a[i+1];
  return b;
}
```

Pentru calculul valorii $v = a'(x)$ a derivatei polinomului $a$ în $x$ este suficient apelul

`v=valp(derivp(a),x);`.

Dacă vrem să calculăm derivata de ordinul $k \geq 0$ a polinomului $a$, atunci

```
static int[] derivpk(int[] a,int k)
{
  int i;
  int[] b;
  m=a.length-1;
  b=new int[m+1];
  for(i=0;i<=n;i++)
    b[i]=a[i];
  for(i=1;i<=k;i++)
    b=derivp(b);
  return b;
}
```

Pentru calculul valorii $v = a^{(k)}(x)$ a derivatei de ordinul $k$ a polinomului $a$ în $x$ este suficient apelul

`v=valp(derivpk(a,k),x);`.

## 7.4 Operaţii cu mulţimi

O mulţime $A$ se poate memora într-un vector **a**, ale cărui elemente sunt distincte. Folosind vectorii putem descrie operaţiile cu mulţimi.

### 7.4.1 Apartenenţa la mulţime

Testul de apartenenţă a unui element $x$ la o multime $A$, este prezentat în algoritmul următor:

```
static boolean apartine(int[] a, int x)
{
  int i,n=a.length;
  boolean ap=false;
  for(i=0;i<n;i++)
    if(a[i]==x) {ap=true; break;}
  return ap;
}
```

### 7.4.2 Diferenţa a două mulţimi

Diferenţa a două mulţimi este dată de mulţimea

$$C = A - B = \{x | x \in A, x \notin B\}$$

Notăm $card\ A = m$.

```
static int[] diferenta(int[] a, int[] b)
{
  int i, j=0, m=a.length;
  int[] c=new int[m];
  for(i=0;i<m;i++)
    if(!apartine(b,a[i]) c[j++]=a[i];
  if(j==m) return c;
    else
    {
      int[] cc=new int[j];
      for(i=0;i<j;i++) cc[i]=c[i];
      return cc;
    }
}
```

### 7.4.3 Reuniunea şi intersecţia a două mulţimi

Reuniunea a două mulţimi este multimea:

$$C = A \cup B = A \cup (B - A).$$

Introducem în $C$ toate elementele lui $A$ şi apoi elementele lui $B - A$.

```
static int[] reuniune(int[] a, int[] b)
{
  int i, j, m=a.length, n=b.length;
  int[] c=new int[m+n];
  for(i=0;i<m;i++) c[i]=a[i];
  j=m;
  for(i=0;i<n;i++) if(!apartine(a,b[i]) c[j++]=b[i];
  if(j==m+n) return c;
    else
    {
      int[] cc=new int[j];
      for(i=0;i<j;i++) cc[i]=c[i];
      return cc;
    }
}
```

Intersecţia a două mulţimi este multimea:

$$C = A \cap B = \{x | x \in A \text{ şi } x \in B\}$$

```
static int[] reuniune(int[] a, int[] b)
{
  int i, j, m=a.length;
  int[] c=new int[m];
  j=0;
  for(i=0;i<m;i++) if(apartine(b,a[i]) c[j++]=a[i];
  if(j==m) return c;
    else
    {
      int[] cc=new int[j];
      for(i=0;i<j;i++) cc[i]=c[i];
      return cc;
    }
}
```

### 7.4.4 Produsul cartezian a două mulţimi

Produs cartezian a doua multimi este multimea:

$$A \times B = \{(x, y) | x \in A \text{ şi } y \in B\}$$

Putem stoca produsul cartezian sub forma unei matrice $C$ cu două linii şi $m \times n$ coloane. Fiecare coloană a matricei conţine câte un element al produsului cartezian.

```
static int[][] prodc(int[] a, int[] b)
{
  int i, j, k, m=a.length, n=b.length;
  int[][] c=new int[2][m*n];
  k=0;
  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
    {
      c[0][k]=a[i];
      c[1][k]=b[j];
      k++;
    }
  return c;
}
```

De exemplu, pentru $A = \{1, 2, 3, 4\}$ şi $B = \{1, 2, 3\}$, matricea $C$ este

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| linia 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 |
| linia 1 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |

### 7.4.5 Generarea submulţimilor unei mulţimi

Generarea submulţimilor unei multimi $A = \{a_1, a_2, ..., a_n\}$, este identică cu generarea submulţimilor mulţimii de indici $\{1, 2, ..., n\}$.

O submulţime se poate memora sub forma unui vector cu $n$ componente, unde fiecare componentă poate avea valori 0 sau 1. Componenta $i$ are valoarea 1 dacă elementul $a_i$ aparţine submulţimii şi 0 în caz contrar. O astfel de reprezentare se numeşte *reprezentare prin vector caracteristic*.

Generarea tuturor submulţimilor înseamnă generarea tuturor combinaţiilor de 0 şi 1 care pot fi reţinute de vectorul caracteristic $V$, adică a tuturor numerelor în baza 2 care se pot reprezenta folosind $n$ cifre.

Pentru a genera adunarea în binar, ţinem cont că trecerea de la un ordin la următorul se face când se obţine suma egală cu 2, adică $1 + 1 = (10)_2$.

De exemplu, pentru $n = 4$, vom folosi un vector $v$

| poziţia | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| valoarea | · | · | · | · |

| | | | | | |
|---|---|---|---|---|---|
| iniţial | 0 | 0 | 0 | 0 | şi adunăm 1 |
| obţinem | 0 | 0 | 0 | 1 | şi adunăm 1 |
| obţinem | 0 | 0 | 0 | 2 | care nu este permis, şi trecem la ordinul următor |
| obţinem | 0 | 0 | 1 | 0 | şi adunăm 1 |
| obţinem | 0 | 0 | 1 | 1 | şi adunăm 1 |
| obţinem | 0 | 0 | 1 | 2 | care nu este permis, şi trecem la ordinul următor |
| obţinem | 0 | 0 | 2 | 0 | care nu este permis, şi trecem la ordinul următor |
| obţinem | 0 | 1 | 0 | 0 | şi aşa mai departe |
| obţinem | · | · | · | · | până când |
| obţinem | 1 | 1 | 1 | 1 | |

Aceste rezultate se pot reţine într-o matrice cu $n$ linii şi $2^n$ coloane.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $a_1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| $a_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| $a_3$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 2 |
| $a_4$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 3 |

Ultima coloană conţine numărul liniei din matrice. Coloana 0 reprezintă mulţimea vidă, coloana F reprezintă întreaga mulţime, şi, de exemplu, coloana 5 reprezintă submultimea $\{a_2, a_4\}$ iar coloana 7 reprezintă submultimea $\{a_2, a_3, a_4\}$.

```
static int[][] submultimi(int n)
{
  int i, j, nc=1;
  int[] v=new int[n+1];
  int[][] c;
  for(i=1;i<=n;i++) nc*=2;
  c=new int[n][nc];
  for(i=1;i<=n;i++) v[i]=0;
  j=0;
  while(j<nc)
  {
    v[n]=v[n]+1;
    i=n;
    while(v[i]>1) { v[i]=v[i]-2; v[i-1]=v[i-1]+1; i--; }
    for(i=1;i<=n;i++) c[j][i-1]=v[i];
    j++;
  }
  return c;
}
```

## 7.5 Operaţii cu numere întregi mari

Operaţiile aritmetice sunt definite numai pentru numere reprezentate pe 16, 32 sau 64 biţi. Dacă numerele sunt mai mari, operaţiile trebuie implementate de utilizator.

### 7.5.1 Adunarea şi scăderea

Adunarea şi scăderea sunt directe: aplicând metodele din şcoala elementară.

```
static int[] suma(int[] x, int[] y)
{
  int nx=x.length;
  int ny=y.length;
  int nz;
  if(nx>ny)
    nz=nx+1;
  else
    nz=ny+1;
  int[] z=new int[nz];
  int t,s,i;
  t=0;
  for (i=0;i<=nz-1;i++)
  {
    s=t;
    if(i<=nx-1)
      s=s+x[i];
    if(i<=ny-1)
      s=s+y[i];
    z[i]=s%10;
    t=s/10;
  }
  if(z[nz-1]!=0)
    return z;
  else
  {
    int[] zz=new int[nz-1];
    for (i=0;i<=nz-2;i++) zz[i]=z[i];
    return zz;
  }
}
```

### 7.5.2 Inmulţirea şi împărtirea

Metoda învăţată în şoală este corectă.

```
static int[] produs(int[]x,int[]y)
{
  int nx=x.length;
  int ny=y.length;
  int nz=nx+ny;
  int[] z=new int[nz];
  int[] [] a=new int[ny][nx+ny];
  int i,j;
  int t,s;
  for(j=0;j<=ny-1;j++)
  {
    t=0;
    for(i=0;i<=nx-1;i++)
    {
      s=t+y[j]*x[i];
      a[j][i+j]=s%10;
      t=s/10;
    }
    a[j][i+j]=t;
  }
  t=0;
  for(j=0;j<=nz-1;j++)
  {
    s=0;
    for(i=0;i<=ny-1;i++)
      s=s+a[i][j];
    s=s+t;
    z[j]=s%10;
    t=s/10;
  }
  if(z[nz-1]!=0)
    return z;
  else
  {
    int[] zz=new int [nz-1];
    for(j=0;j<=nz-2;j++)
      zz[j]=z[j];
    return zz;
  }
}
```

### 7.5.3 Puterea

Presupunem că vrem să calculăm $x^n$. Cum facem acest lucru? Este evident că următoarea secvenţă funcţionează:

```
for (p = 1, i = 0; i < n; i++) p *= x;
```

Presupunând că toate înmulţirile sunt efectuate într-o unitate de timp, acest algoritm are complexitatea $O(n)$. Totuşi, putem să facem acest lucru mai repede! Presupunând, pentru început, că $n = 2^k$, următorul algoritm este corect:

```
for (p = x, i = 1; i < n; i *= 2) p *= p;
```

Aici numărul de treceri prin ciclu este egal cu $k = \log_2 n$.

Acum, să considerăm cazul general. Presupunem că $n$ are expresia binară $(b_k, b_{k-1}, ..., b_1, b_0)$. Atunci putem scrie

$$n = \sum_{i=0, b_i=1}^{k} 2^i.$$

Deci,

$$x^n = \prod_{i=0, b_i=1}^{k} x^{2^i}.$$

```
int exponent_1(int x, int n)
{
  int c, z;
  for (c = x, z = 1; n != 0; n = n / 2)
  {
    if (n & 1) /* n este impar */
      z *= c;
    c *= c;
  }
  return z;
}

int exponent_2(int x, int n)
{
  if (n == 0)
    return 1;
  if (n & 1) /* n este impar */
    return x * exponent_2(x, n - 1);
  return exponent_2(x, n / 2) * exponent_2(x, n / 2);
}
```

```
int exponent_3(int x, int n)
{
  int y;
  if (n == 0)
    return 1;
  if (n & 1) /* n este impar */
    return x * exponent_3(x, n - 1);
  y = exponent_3(x, n / 2);
  return y * y;
}
```

## 7.6 Operaţii cu matrice

### 7.6.1 Înmulţirea

O funcţie scrisă în C/C++:

```
void matrix_product(int** A, int** B, int** C)
{
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
      C[i][j] = 0;
      for (k = 0; k < n; k++)
        C[i][j] += A[i][k] * B[k][j];
    }
}
```

### 7.6.2 Inversa unei matrice

O posibilitate este cea din şcoală. Aceasta presupune calculul unor determinanţi. Determinantul $det(A)$ se defineşte recursiv astfel:

$$\det(A) = \sum_{i=0}^{n-1} (-1)^{i+j} * a_{i,j} * \det(A_{i,j}).$$

unde $a_{i,j}$ este element al matricei iar $A_{i,j}$ este submatricea obţinută prin eliminarea liniei $i$ şi a coloanei $j$.

```
int determinant(int n, int[][] a)
{
  if (n == 1)
    return a[0][0];
  int det = 0;
  int sign = 1;
  int[][] b = new int[n - 1][n - 1];
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < i; j++)
      for (int k = 1; k < n; k++)
        b[j][k - 1] = a[j][k];
    for (int j = i + 1; j < n; j++)
      for (int k = 1; k < n; k++)
        b[j - 1][k - 1] = a[j][k];
    det += sign * a[i][0] * determinant(n - 1, b);
    sign *= -1;
  }
}
```

Folosind determinanţi, inversa matricei se poate calcula folosind *regula lui Cramer*. Presupunem că $A$ este inversabilă şi fie $B = (b_{i,j})$ matricea definită prin

$$b_{i,j} = (-1)^{i+j} * \det(A_{i,j}) / \det(A).$$

Atunci $A^{-1} = B^T$, unde $B^T$ este transpusa matricei $B$.

# Capitolul 8

# Algoritmi combinatoriali

## 8.1 Principiul includerii şi al excluderii şi aplicaţii

### 8.1.1 Principiul includerii şi al excluderii

Fie $A$ şi $B$ două mulţimi finite. Notăm prin $|A|$ cardinalul mulţimii $A$. Se deduce uşor că:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Fie $A$ o mulţime finită şi $A_1, A_2, ..., A_n$ submulţimi ale sale. Atunci numărul elementelor lui $A$ care nu apar în nici una din submulţimile $A_i$ $(i = 1, 2, ..., n)$ este egal cu:

$$|A| - \sum_{i=1}^{n} |A_i| + \sum_{1 \le i < j \le n} |A_i \cap A_j| - \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_k| + ... + (-1)^n |A_1 \cap A_2 \cap ... \cap A_n|$$

Se pot demonstra prin inducţie matematică următoarele formule:

$$|\bigcup_{i=1}^{n} A_i| = \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| + \sum_{1 \le i < j < k \le n} |A_i \cap A_j \cap A_j| - ... + (-1)^{n+1} |\bigcap_{i=1}^{n} A_i|$$

$$|\bigcap_{i=1}^{n} A_i| = \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cup A_j| + \sum_{1 \le i < j < k \le n} |A_i \cup A_j \cup A_j| - ... + (-1)^{n+1} |\bigcup_{i=1}^{n} A_i|$$

### 8.1.2 Determinarea funcţiei lui Euler

Funcţia $\phi(n)$ a lui Euler ne dă numărul numerelor naturale mai mici ca $n$ şi prime cu $n$.

Numărul $n$ poate fi descompus în factori primi sub forma:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} ... p_m^{\alpha_m}$$

Notăm cu $A_i$ mulţimea numerelor naturale mai mici ca $n$ care sunt multipli de $p_i$. Atunci avem:

$$|A_i| = \frac{n}{p_i}, |A_i \cap A_j| = \frac{n}{p_i p_j}, |A_i \cap A_j \cap A_k| = \frac{n}{p_i p_j p_k}, ...$$

Rezultă:

$$\phi(n) = n - \sum_{i=1}^{m} \frac{n}{p_i} + \sum_{1 \le i < j \le m} \frac{n}{p_i p_j} - \sum_{1 \le i < j < k \le m} \frac{n}{p_i p_j p_k} + ... + (-1)^m \frac{n}{p_1 p_2 ... p_m}$$

care este tocmai dezvoltarea produsului

$$\phi(n) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) ... \left(1 - \frac{1}{p_m}\right)$$

```
class Euler
{
  static long[] fact;
  public static void main (String[]args)
  {
    long n=36L; // Long.MAX_VALUE=9.223.372.036.854.775.807;
    long nrez=n;
    long[] pfact=factori(n);
    // afisv(fact);
    // afisv(pfact);
    int k,m=fact.length-1;
    for(k=1;k<=m;k++) n/=fact[k];
    for(k=1;k<=m;k++) n*=fact[k]-1;
    System.out.println("f("+nrez+") = "+n);
  }

  static long[] factori(long nr)
  {
    long d, nrrez=nr;
    int nfd=0; // nr. factori distincti
    boolean gasit=false;
```

```
    while((nr!=1)&(nr%d==0)) { nr=nr/d; gasit=true; }
    if(gasit) {nfd++;gasit=false;}
    d=3;
    while(nr!=1)
    {
      while((nr!=1)&(nr%d==0)) { nr=nr/d; gasit=true; }
      if(gasit) {nfd++;gasit=false;}
      d=d+2;
    }
    nr=nrrez;
    fact=new long[nfd+1];
    long[] pf=new long[nfd+1];
    int pfc=0; // puterea factorului curent
    nfd=0;   // nr. factori distincti
    gasit=false;
    d=2;
    while((nr!=1)&(nr%d==0)) { nr=nr/d; gasit=true; pfc++; }
    if(gasit) {fact[++nfd]=d;pf[nfd]=pfc;gasit=false;pfc=0;}
    d=3;
    while(nr!=1)
    {
      while((nr!=1)&(nr%d==0)) { nr=nr/d;  gasit=true; pfc++; }
      if(gasit) {fact[++nfd]=d;pf[nfd]=pfc;gasit=false;pfc=0;}
      d=d+2;
    }
    return pf;
  }//descfact

  static void afisv(long[] a)
  {
    for(int i=1;i<a.length;i++) System.out.print(a[i]+" ");
    System.out.println();
  }
}
```

### 8.1.3 Numărul funcţiilor surjective

Se dau mulţimile $X = \{x_1, x_2, ..., x_m\}$ şi $Y = \{y_1, y_2, ..., y_n\}$.

Fie $S_{m,n}$ numărul funcţiilor surjective $f : X \longrightarrow Y$.

Fie $A = \{f | f : X \longrightarrow Y\}$ (mulţimea tuturor funcţiilor definite pe $X$ cu valori în $Y$) şi $A_i = \{f | f : X \longrightarrow Y, y_i \notin f(X)\}$ (mulţimea funcţiilor pentru care $y_i$ nu este imaginea nici unui element din $X$).

Atunci

$$S_{m,n} = |A| - |\bigcup_{i=1}^{n} A_i|$$

Folosind principiul includerii şi al excluderii, obţinem

$$S_{m,n} = |A| - \sum_{i=1}^{n} |A_i| + \sum_{1 \le i < j \le n} |A_i \cap A_j| - ... + (-1)^n |A_1 \cap A_2 \cap ... \cap A_n|$$

Se poate observa uşor că $|A| = n^m$, $|A_i| = (n-1)^m$, $|A_i \cap A_j| = (n-2)^m$, etc.

Din $Y$ putem elimina $k$ elemente în $C_n^k$ moduri, deci

$$\sum_{1 \le i_1 < i_2 < ... < i_k \le n} |\bigcap_{j=1}^{k} A_{i_j}| = C_n^k (n-k)^m$$

Rezultă:

$$S_{m,n} = n^m - C_n^1 (n-1)^m + C_n^2 (n-2)^m + ... + (-1)^{n-1} C_n^{n-1}$$

Observaţii:

1. Deoarece $A_1 \cap A_2 \cap ... \cap A_n = \emptyset$ şi pentru că nu poate exista o funcţie care să nu ia nici o valoare, ultimul termen lipseşte.

2. Dacă $n = m$ atunci numărul funcţiilor surjective este egal cu cel al funcţiilor injective, deci $S_{m,n} = n!$ şi se obţine o formulă interesantă:

$$n! = \sum_{k=0}^{n-1} (-1)^k C_n^k (n-k)^n$$

```
class Surjectii
{
  public static void main (String[]args)
  {
    int m, n=5, k, s;
    for(m=2;m<=10;m++)
    {
      s=0;
      for(k=0;k<=n-1;k++)
        s=s+comb(n,k)*putere(-1,k)*putere(n-k,m);
      System.out.println(m+" : "+s);
    }
    System.out.println("GATA");
  }
```

```
  static int putere (int a, int n)
  {
    int rez=1, k;
    for(k=1;k<=n;k++) rez=rez*a;
    return rez;
  }

  static int comb (int n, int k)
  {
    int rez, i, j, d;
    int[] x=new int[k+1];
    int[] y=new int[k+1];
    for(i=1;i<=k;i++) x[i]=n-k+i;
    for(j=1;j<=k;j++) y[j]=j;
    for(j=2;j<=k;j++)
      for(i=1;i<=k;i++)
      {
        d=cmmdc(y[j],x[i]);
        y[j]=y[j]/d;
        x[i]=x[i]/d;
        if(y[j]==1) break;
      }
    rez=1;
    for(i=1;i<=k;i++) rez=rez*x[i];
    return rez;
  }

  static int cmmdc (int a,int b)
  {
    int d,i,c,r;
    if (a>b) {d=a;i=b;} else{d=b;i=a;}
    while (i!=0) { c=d/i; r=d%i; d=i; i=r; }
    return d;
  }
}
```

### 8.1.4 Numărul permutărilor fără puncte fixe

Fie $X = \{1, 2, ..., n\}$. Dacă $p$ este o permutare a elementelor mulţimii $X$, spunem că numărul $i$ este un punct fix al permutării $p$, dacă $p(i) = i$ $(1 \leq i \leq n)$.

Se cere să se determine numărul $D(n)$ al permutărilor fără puncte fixe, ale mulţimii $X$. Să notăm cu $A_i$ mulţimea celor $(n-1)!$ permutări care admit un punct fix în $i$ (dar nu obligatoriu numai acest punct fix!). Folosind principiul includerii

şi al excluderii, numărul permutărilor care admit cel puţin un punct fix este egal cu:

$$|A_1 \cup A_2 \cup ... \cup A_n| = \sum_{i=1}^{n} |A_i| - \sum_{1 \le i < j \le n} |A_i \cap A_j| + ... + (-1)^{n-1} |\bigcap_{i=1}^{n} A_i|.$$

Dar

$$|A_{i_1} \cap A_{i_2} \cap ... \cap A_{i_k}| = (n-k)!$$

deoarece o permutare din mulţimea $A_{i_1} \cap A_{i_2} \cap ... \cap A_{i_k}$ are puncte fixe în poziţiile $i_1, i_2, ..., i_k$, celelalte poziţii conţinând o permutare a celor $n-k$ elemente rămase (care pot avea sau nu puncte fixe!). Cele $k$ poziţii $i_1, i_2, ..., i_k$ pot fi alese în $C_n^k$ moduri, deci

$$|A_1 \cup A_2 \cup ... \cup A_n| = C_n^1 (n-1)! - C_n^2 (n-2)! + ... + (-1)^{n-1} C_n^n.$$

Atunci

$$\begin{aligned} D(n) &= n! - |A_1 \cup A_2 \cup ... \cup A_n| = \\ &= n! - C_n^1 (n-1)! + C_n^2 (n-2)! - + ... + (-1)^n C_n^n \\ &= n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + ... + \frac{(-1)^n}{n!}\right). \end{aligned}$$

De aici rezultă că

$$\lim_{n \to \infty} \frac{D(n)}{n!} = e^{-1},$$

deci, pentru $n$ mare, probabilitatea ca o permutare a $n$ elemente, aleasă aleator, să nu aibă puncte fixe, este de $e^{-1} \approx 0.3678$.

Se poate demonstra uşor că:

$$\begin{aligned} D(n+1) &= (n+1) D(n) + (-1)^{n+1} \\ D(n+1) &= n \left(D(n) + D(n-1)\right). \end{aligned}$$

```
class PermutariFixe
{
  public static void main(String [] args)
  {
    long n=10,k,s=0L,xv,xn; // n=22 maxim pe long !
    if((n&1)==1) xv=-1L; else xv=1L;
    s=xv;
    for(k=n;k>=3;k--) { xn=-k*xv; s+=xn; xv=xn; }
    System.out.println("f("+n+") =  "+s);
  }
}
```

## 8.2 Principiul cutiei lui Dirichlet şi aplicaţii

Acest principiu a fost formulat prima dată de Dirichle (1805-1859).
În forma cea mai simplă acest principiu se enunţă astfel:

*Dacă n obiecte trebuie împărţite în mai puţin de n mulţimi, atunci există cel puţin o mulţime în care vor fi cel puţin două obiecte.*

Mai general, principiul lui Dirichlet se poate enunţa astfel:

*Fiind date m obiecte, care trebuie împărţite în n mulţimi, şi un număr natural k astfel încât $m > kn$, atunci, în cazul oricărei împărţiri, va exista cel puţin o mulţime cu cel puţin $k+1$ obiecte.*

Pentru $k = 1$ se obţine formularea anterioară.
Cu ajutorul funcţiilor, principiul cutiei se poate formula astfel:

Fie $A$ şi $B$ două mulţimi finite cu $|A| > |B|$ şi funcţia $f : A \longrightarrow B$. Atunci, există $b \in B$ cu proprietatea că $|f^{-1}(b)| \geq 2$. Dacă notăm $|A| = n$ şi $|B| = r$ atunci $|f^{-1}(b)| \geq \left\lfloor \frac{n}{r} \right\rfloor$.

Demonstrăm ultima inegalitate. Dacă aceasta nu ar fi adevărată, atunci

$$|f^{-1}(b)| < \left\lfloor \frac{n}{r} \right\rfloor, \forall b \in B.$$

Dar mulţimea $B$ are $r$ elemente, deci

$$n = \sum_{b \in B} |f^{-1}(b)| < r \cdot \frac{n}{r} = n$$

ceea ce este o contradicţie.

### 8.2.1 Problema subsecvenţei

Se dă un şir finit $a_1, a_2, ..., a_n$ de numere întregi. Există o subsecvenţă $a_i, a_{i+1}, ..., a_j$ cu proprietatea că $a_i + a_{i+1} + ... + a_j$ este un multiplu de $n$.

Să considerăm următoarele sume:

$$\begin{aligned} s_1 &= a_1, \\ s_2 &= a_1 + a_2, \\ &... \\ s_n &= a_1 + a_2 + ... + a_n. \end{aligned}$$

Dacă există un $k$ astfel $s_k$ este multiplu de $n$ atunci $i = 1$ şi $j = k$.

Dacă nici o sumă parţială $s_k$ nu este multiplu de $n$, atunci resturile împărţirii acestor sume parţiale la $n$ nu pot fi decât în mulţimea $\{1, 2, ..., n-1\}$. Pentru că avem $n$ sume parţiale şi numai $n-1$ resturi, înseamnă că există cel puţin două sume parţiale ($s_{k_1}$ şi $s_{k_2}$, unde $k_1 < k_2$) cu acelaşi rest. Atunci subsecvenţa căutată se obţine luând $i = k_1 + 1$ şi $j = k_2$.

### 8.2.2 Problema subşirurilor strict monotone

Se dă şirul de numere reale distincte $a_1, a_2, ..., a_{mn+1}$. Atunci, şirul conţine un subşir crescător de $m+1$ elemente:

$$a_{i_1} < a_{i_2} < ... < a_{i_{m+1}} \text{ unde } 1 \le i_1 < i_2 < ... < i_{m+1} \le mn+1,$$

sau un subşir descrescător de $n+1$ elemente

$$a_{j_1} < a_{j_2} < ... < a_{j_{n+1}} \text{ unde } 1 \le j_1 < j_2 < ... < j_{n+1} \le mn+1,$$

sau ambele tipuri de subşiruri.

Fiecărui element al şirului îi asociem perechea de numere naturale $(x_i, y_i)$ unde $x_i$ este lungimea maximă a subşirurilor crescătoare care încep cu $a_i$ iar $y_i$ este lungimea maximă a subşirurilor descrescătoare care încep în $a_i$.

Presupunem că afirmaţia problemei nu este adevărată, adică: pentru toate numerele naturale $x_i$ şi $y_i$ avem $1 \le x_i \le m$ şi $1 \le y_i \le n$. Atunci perechile de numere $(x_i, y_i)$ pot avea $mn$ elemente distincte.

Deoarece şirul are $mn+1$ termeni, există un $a_i$ şi un $a_j$ pentru care perechile de numere $(x_i, y_i)$ şi $(x_j, y_j)$ sunt identice ($x_i = x_j$, $y_i = y_j$), dar acest lucru este imposibil (cei doi termeni $a_i$ şi $a_j$ ar trebui să coincidă), ceea ce este o contradiţie.

Deci există un subşir crescător cu $m+1$ termeni sau un subşir descrescător cu $n+1$ termeni.

## 8.3 Numere remarcabile

### 8.3.1 Numerele lui Fibonacci

Numerele lui Fibonacci se pot defini recursiv prin:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ pentru } n \ge 2. \tag{8.3.1}$$

Primele numere Fibonacci sunt:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...$$

Se poate arăta că

$$F_n = \frac{1}{2^n\sqrt{5}}\left(\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n\right).$$

Numerele lui Fibonacci satisfac multe identităţi interesante, ca de exemplu:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \quad (8.3.2)$$

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n \quad (8.3.3)$$

$$F_{n+m} = F_mF_{n+1} + F_{m-1}F_n \quad (8.3.4)$$

$$F_{nk} = \text{ multiplu de } F_k \quad (8.3.5)$$

$$(8.3.6)$$

şi

$$F_2 + F_4 + ... + F_{2n} = F_{2n+1} - 1 \quad (8.3.7)$$

$$F_1 + F_3 + ... + F_{2n-1} = F_{2n} \quad (8.3.8)$$

$$F_1^2 + F_2^2 + ... + F_n^2 = F_nF_{n+1} \quad (8.3.9)$$

$$F_1F_2 + F_2F_3 + ... + F_{2n-1}F_{2n} = F_{2n}^2 \quad (8.3.10)$$

$$F_1F_2 + F_2F_3 + ... + F_{2n}F_{2n+1} = F_{2n+1}^2 - 1 \quad (8.3.11)$$

**Teorema 2** *Orice număr natural n se poate descompune într-o sumă de numere Fibonacci. Dacă nu se folosesc în descompunere numerele $F_0$ şi $F_1$ şi nici două numere Fibonacci consecutive, atunci această descompunere este unică abstracţie făcând de ordinea termenilor.*

Folosind această descompunere, numerele naturale pot fi reprezentate asemănător reprezentării în baza 2. De exemplu

$$19 = 1 \cdot 13 + 0 \cdot 8 + 1 \cdot 5 + 0 \cdot 3 + 0 \cdot 2 + 1 \cdot 1 = (101001)_F$$

În această scriere nu pot exista două cifre 1 alăturate.

```
import java.io.*;
class DescFibo
{
  static int n=92;
  static long[] f=new long[n+1];

  public static void main (String[]args) throws IOException
  {
    int iy, k, nrt=0;
    long x,y; // x=1234567890123456789L; cel mult!
```

```
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    System.out.print("x = ");
    x=Long.parseLong(br.readLine());
    f[0]=0;
    f[1]=1;
    f[2]=1;
    for(k=3;k<=n;k++) f[k]=f[k-1]+f[k-2];
    for(k=0;k<=n;k++) System.out.println(k+" : "+f[k]);
    System.out.println("     "+Long.MAX_VALUE+" = Long.MAX_VALUE");
    System.out.println(" x = "+x);
    while(x>0)
    {
      iy=maxFibo(x);
      y=f[iy];
      nrt++;
      System.out.println(nrt+" : "+x+" f["+iy+"] = "+y);
      x=x-y;
    }
  }

  static int maxFibo(long nr)
  {
    int k;
    for(k=1;k<=n;k++) if (f[k]>nr) break;
    return k-1;
  }
}
```

### 8.3.2 Numerele lui Catalan

Numerele

$$C_n = \frac{1}{n+1} C_{2n}^n$$

se numesc numerele lui Catalan. Ele apar în multe probleme, ca de exemplu: numărul arborilor binari, numărul de parantezări corecte, numărul drumurilor sub diagonală care unesc punctele $(0,0)$ şi $(n,n)$ formate din segmente orizontale şi verticale, numărul secvenţelor cu $n$ biţi în care numărul cifrelor 1 nu depăşeşte numărul cifrelor 0 în nici o poziţie plecând de la stânga spre dreapta, numărul segmentelor care unesc $2n$ puncte în plan fără să se intersecteze, numărul şirurilor $(x_1, x_2, ..., x_{2n})$ în care $x_i \in \{-1, 1\}$ şi $x_1 + x_2 + ... + x_{2n} = 0$ cu proprietatea $x_1 + x_2 + ... + x_i \geq 0$ pentru orice $i = 1, 2, ..., 2n-1$, numărul modurilor de a triangulariza un poligon, şi multe altele.

Numerele lui Catalan sunt soluţie a următoarei ecuaţii de recurenţă:

$$C_{n+1} = C_0C_n + C_1C_{n-1} + ... + C_nC_0, \text{ pentru } n \geq 0 \text{ şi } C_0 = 1.$$

Numerele lui Catalan verifică şi relaţia:

$$C_{n+1} = \frac{4n+2}{n+2}C_n$$

O implementare cu numere mari este:

```
class Catalan
{
  public static void main (String[]args)
  {
    int n;
    int[] x;
    for(n=1;n<=10;n++)
    {
      x=Catalan(n);
      System.out.print(n+" : ");
      afisv(x);
    }
  }

  static int[] inm(int[]x,int[]y)
  {
    int i, j, t, n=x.length, m=y.length;
    int[][]a=new int[m][n+m];
    int[]z=new int[m+n];
    for(j=0;j<m;j++)
    {
      t=0;
      for(i=0;i<n;i++)
      {
        a[j][i+j]=y[j]*x[i]+t;
        t=a[j][i+j]/10;
        a[j][i+j]=a[j][i+j]%10;
      }
      a[j][i+j]=t;
    }
    t=0;
    for(j=0;j<m+n;j++)
    {
      z[j]=t;
      for(i=0;i<m;i++) z[j]=z[j]+a[i][j];
```

```
    t=z[j]/10;
    z[j]=z[j]%10;
  }
  if(z[m+n-1]!= 0) return z;
  else
  {
    int[]zz=new int[m+n-1];
    for(i=0;i<=m+n-2;i++) zz[i]=z[i];
    return zz;
  }
}

static void afisv(int[]x)
{
  int i;
  for(i=x.length-1;i>=0;i--) System.out.print(x[i]);
  System.out.print(" *** "+x.length);
  System.out.println();
}

static int[] nrv(int nr)
{
  int nrrez=nr;
  int nc=0;
  while(nr!=0) { nc++; nr=nr/10; }
  int[]x=new int [nc];
  nr=nrrez;
  nc=0;
  while(nr!=0) { x[nc]=nr%10; nc++; nr=nr/10; }
  return x;
}

static int[] Catalan(int n)
{
  int[] rez;
  int i, j, d;
  int[] x=new int[n+1];
  int[] y=new int[n+1];
  for(i=2;i<=n;i++) x[i]=n+i;
  for(j=2;j<=n;j++) y[j]=j;
  for(j=2;j<=n;j++)
    for(i=2;i<=n;i++)
    {
      d=cmmdc(y[j],x[i]);
```

```
        y[j]=y[j]/d;
        x[i]=x[i]/d;
        if(y[j]==1) break;
      }
    rez=nrv(1);
    for(i=2;i<=n;i++) rez=inm(rez,nrv(x[i]));
    return rez;
  }

  static int cmmdc (int a,int b)
  {
    int d,i,c,r;
    if (a>b) {d=a;i=b;} else{d=b;i=a;}
    while (i!=0) { c=d/i; r=d%i; d=i; i=r; }
    return d;
  }
}
```

## 8.4 Probleme rezolvate

1. Să se determine numărul arborilor binari cu $n$ vârfuri.

*Rezolvare*: Fie $b(n)$ numărul arborilor binari cu $n$ vârfuri. Prin convenţie $b_0 = 1$. Prin desene $b_1 = 1$, $b_2 = 2$, $b_3 = 5$, şi: dacă fixăm rădăcina arborelui, ne mai rămân $n-1$ vârfuri care pot apărea în subarborele stâng sau drept; dacă în subarborele stâng sunt $k$ vârfuri, în subarborele drept trebuie să fie $n-1-k$ vârfuri; cu aceşti subarbori se pot forma în total $b_k b_{n-1-k}$ arbori; adunând aceste valori pentru $k = 0, 1, ..., n-1$ vom obţine valoarea lui $b_n$. Deci, pentru $n \geq 1$

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + ... + b_{n-1} b_0 = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$

Se obţine

$$b_n = \frac{1}{n+1} C_{2n}^n$$

2. Care este numărul permutărilor a $n$ obiecte cu $p$ puncte fixe?

*Rezolvare*: Deoarece cele $p$ puncte fixe pot fi alese în $C_n^p$ moduri, şi cele $n-p$ puncte rămase nu mai sunt puncte fixe pentru permutare, rezultă că numărul permutărilor cu $p$ puncte fixe este egal cu

$$C_n^p D(n-p)$$

deoarece pentru fiecare alegere a celor $p$ puncte fixe există $D(n-p)$ permutări ale obiectelor rămase, fără puncte fixe.

3. Fie $X = \{1, 2, ..., n\}$. Dacă $E(n)$ reprezintă numărul permutărilor pare[1] ale mulţimii $X$ fără puncte fixe, atunci

$$E(n) = \frac{1}{2}\left(D(n) + (-1)^{n-1}(n-1)\right).$$

*Rezolvare:* Fie $A_i$ mulţimea permutărilor pare $p$ astfel încât $p(i) = i$. Deoarece numărul permutărilor pare este $\frac{1}{2}n!$, rezultă că

$$\begin{aligned} E(n) &= \frac{1}{2}n! - |A_1 \cup ... \cup A_n| = \\ &= \frac{1}{2}n! - C_n^1(n-1)! + C_n^2(n-2)! + ... + (-1)^n C_n^n. \end{aligned}$$

Ţinând cont că

$$|A_{i_1} \cap A_{i_2} \cap ... \cap A_{i_k}| = (n-k)!$$

rezultă formula cerută.

[1]Dacă $i < j$, şi în permutare $i$ apare după $j$, spunem că avem o *inversiune*. O permutare este *pară* dacă are un număr par de inversiuni

# Capitolul 9

# Algoritmi de căutare

## 9.1 Problema căutării

Problema căutării este: se dă un vector $a$ cu $n$ elemente şi o valoare $x$ de acelaşi tip cu elementele din $a$. Să se determine $p$ astfel încât $x = a[p]$ sau $-1$ dacă nu există un element cu valoarea $v$ în $a$.

O tabelă cu câmpuri care nu sunt de acelaşi tip se poate organiza cu ajutorul vectorilor dacă numărul de coloane este suficient de mic. De exemplu, o tabelă cu trei informaţii: număr curent, nume, telefon poate fi organizată cu ajutorul a doi vectori (nume şi telefon) iar numărul curent este indicele din vector.

## 9.2 Căutarea secvenţială

```
static int cauta (String x) {
  for (int i = 0; i < N; ++i)
    if (x.equals(nume[i])) return telefon[i];
  return 1;
}
```

se poate scrie şi sub forma:

```
static int cauta (String x) {
  int i = 0;
  while (i < N && !x.equals(nume[i])) ++i;
  if (i < N) return telefon[i];
  else return 1;
}
```

O altă posibilitate este de a pune o *santinelă* în capătul tabelei.

```
static int cauta (String x) {
  int i = 0;
  nume[N] = x; telefon[N] = 1;
  while (! x.equals(nume[i])) ++i;
  return tel[i];
}
```

Scrierea procedurii de căutare într-un tabel de nume este în acest caz mai eficientă, pentru că nu se face decât un singur test în plus aici (în loc de două teste). Căutarea secvenţială se mai numeşte şi căutare lineară, pentru că se execută $N/2$ operaţii în medie, şi $N$ operaţii în cazul cel mai defavorabil. Într-un tablou cu 10.000 elemente, căutarea execută 5.000 operaţii în medie, ceea ce înseamnă un consum de timp de aproximativ 0.005 ms (milisecunde).

Iată un program complet care utilizează căutarea lineară într-o tabelă.

```
class Tabela {

  final static int N = 6;
  static String nume[] = new String[N+1];
  static int telefon[] = new int[N+1];

  static void initializare() {
    nume[0] = "Paul";    telefon[0] = 2811;
    nume[1] = "Robert";  telefon[1] = 4501;
    nume[2] = "Laura";   telefon[2] = 2701;
    nume[3] = "Ana";     telefon[3] = 2702;
    nume[4] = "Tudor";   telefon[4] = 2805;
    nume[5] = "Marius";  telefon[5] = 2806;
  }

  static int cauta(String x) {
    for (int i = 0; i < N; ++i)
      if (x.equals(nume[i]))
        return tel[i];
    return 1;
  }

  public static void main (String args[]) {
    initializare();
    if (args.length == 1)
      System.out.println(cauta(args[0]));
  }
}
```

Cel mai simplu algoritm care rezolvă această problemă este *căutarea liniară*, care testează elementele vectorului unul dupa altul, începând cu primul.

```
p=-1;
for(i=0;i<n;i++)
  if(x==a[i]) { p=i; break; }
```

Să analizăm complexitatea acestui algoritm pe cazul cel mai defavorabil, acela în care $v$ nu se găseşte în $a$. În acest caz se va face o parcurgere completă a lui $a$. Considerând ca operaţie elementară testul $v == a[i]$, numărul de astfel de operaţii elementare efectuate va fi egal cu numarul de elemente al lui $a$, adică $n$. Deci complexitatea algoritmului pentru cazul cel mai defavorabil este $O(n)$.

## 9.3 Căutare binară

O altă tehnică de căutare în tabele este căutarea binară. Presupunem că tabela de nume este sortată în ordine alfabetică (cum este cartea de telefoane). În loc de a căuta secvenţial, se compară cheia de căutare cu numele care se află la mijlocul tabelei de nume. Dacă acesta este acelaşi, se returnează numărul de telefon din mijloc, altfel se reîncepe căutarea în prima jumătate (sau în a doua) dacă numele căutat este mai mic (respectiv, mai mare) decât numele din mijlocul tabelei.

```
static int cautareBinara(String x) {
  int i, s, d, cmp;
  s = 0; d = N1;
  do {
    i = (s + d) / 2;
    cmp = x.compareTo(nume[i]);
    if (cmp == 0)
      return telefon[i];
    if (cmp < 0)
      d = i  1;
    else
      s = i + 1;
  } while (s <= d);
  return 1;
}
```

Numărul $C_N$ de comparaţii efectuate pentru o tabelă de dimensiune $N$ este $C_N = 1 + C_{\lfloor N/2 \rfloor}$, unde $C_0 = 1$. Deci $C_N \approx \log_2 N$.

De acum înaninte, $\log_2 N$ va fi scris mai simplu $\log N$.

Dacă tabela are 10.000 elemente, atunci $C_N \approx 14$. Acesta reprezintă un beneficiu considerabil în raport cu 5.000 de operaţii necesare la căutarea lineară.

Desigur căutarea secvenţială este foarte uşor de programat, şi ea se poate folosi pentru tabele de dimensiuni mici. Pentru tabele de dimensiuni mari, căutarea binară este mult mai interesantă.

Se poate obţine un timp sub-logaritmic dacă se cunoaşte distribuţia obiectelor. De exemplu, în cartea de telefon, sau în dicţionar, se ştie apriori că un nume care începe cu litera $V$ se află către sfârşit. Presupunând distribuţia uniformă, putem face o "regulă de trei simplă" pentru găsirea indicelui elementului de referinţă pentru comparare, în loc să alegem mijlocul, şi să urmăm restul algoritmului de căutare binară. Această metodă se numeşte *căutare prin interpolare*. Timpul de căutare este $O(\log \log N)$, ceea ce înseamnă cam 4 operaţii pentru o tabelă de 10.000 elemente şi 5 operaţii pentru $10^9$ elemente în tabelă. Este spectaculos!

O implementare iterativă a căutării binare într-un vector ordonat (crescător sau descrescător) este:

```
int st, dr, m;
boolean gasit;
st=0;
dr=n-1;
gasit=false;
while((st < dr) && !gasit)
{
  m=(st+dr)/2;
  if(am]==x)
    gasit=true;
    else if(a[m] > x)
           dr=m-1;
           else st=m+1;
}
if(gasit) p=m; else p=-1;
```

Algoritmul poate fi descris, foarte elegant, recursiv.

## 9.4 Inserare în tabelă

La căutarea secvenţială sau binară nu am fost preocupaţi de inserarea în tabelă a unui nou element. Aceasta este destul de rară în cazul cărţii de telefon dar în alte aplicaţii, de exemplu lista utilizatorilor unui sistem informatic, este frecvent utilizată.

Vom vedea cum se realizează inserarea unui element într-o tabelă, în cazul căutării secvenţiale şi binare.

Pentru cazul secvenţial este suficient să adăugăm la capătul tabelei noul element, dacă are loc. Dacă nu are loc, se apelează o procedură `error` care afişează un mesaj de eroare.

```
void inserare(String x, int val) {
  ++n;
  if (n >= N)
    error ("Depasire tabela");
  numem[n] = x;
  telefon[n] = val;
}
```

Inserarea se face deci în timp constant, de ordinul $O(1)$.

În cazul căutării binare, trebuie menţinut tabelul ordonat. Pentru inserarea unui element nou în tabelă, trebuie găsită poziţia sa printr-o căutare binară (sau secvenţială), apoi se deplasează toate elementele din spatele ei spre dreapta cu o poziţie pentru a putea insera noul element în locul corect. Aceasta necesită $\log n+n$ opereţii. Inserarea într-o tabelă ordonată cu $n$ elemente este deci de ordinul $O(n)$.

## 9.5 Dispersia

O altă metodă de căutare în tabele este *dispersia*. Se utilizează o funcţie $h$ de grupare a cheilor (adesea şiruri de caractere) într-un interval de numere întregi. Pentru o cheie $x$, $h(x)$ este locul unde se afă $x$ în tabelă. Totul este în ordine dacă $h$ este o aplicaţie injectivă. De asemenea, este bine ca funcţia aleasă să permită evaluarea cu un număr mic de operaţii. O astfel de funcţie este

$$h(x) = \left(x_1 B^{m-1} + x_2 B^{m-2} + ... + x_{m-1} B + x_m\right) \mod N.$$

De obicei se ia $B = 128$ sau $B = 256$ şi se presupune că dimensiunea tabelei $N$ este un număr prim. De ce? Pentru că înmulţirea puterilor lui 2 se poate face foarte uşor prin operaţii de deplasare pe biţi, numerele fiind reprezentate în binar. În general, la maşinile (calculatoarele) moderne, aceste operaţii sunt net mai rapide decât înmulţirea numerelor oarecare. Cât despre alegerea lui $N$, aceasta se face pentru a evita orice interferenţă între între înmulţirile prin $B$ şi împărţirile prin $N$. Într-adevăr, dacă de exemplu $B = N = 256$, atunci $h(x) = x(m)$ este funcţia $h$ care nu depinde decât de ultimul caracter al lui $x$. Scopul este de a avea o funcţie $h$, de dispersie, simplu de calculat şi având o bună distribuţie pe intervalul $[0, N-1]$. Calculul funcţiei $h$ se face prin funcţia `h(x,m)`, unde $m$ este lungimea şirului $x$,

```
static int h(String x){
  int r = 0;
  for (int i = 0; i < x.length(); ++i)
    r = ((r * B) + x.charAt(i)) % N;
  return r;
}
```

Deci funcţia $h$ dă pentru toate cheile $x$ o intrare posibilă în tabelă. Se poate apoi verifica dacă $x = nume[h(x)]$. Dacă da, căutarea este terminată. Dacă nu, înseamnă că tabela conţine o altă cheie astfel încât $h(x') = h(x)$. Se spune atunci că există o *coliziune*, şi tabela trebuie să gestioneze coliziunile. O metodă simplă este de a lista coliziunile într-un tabel `col` paralel cu tabelul `nume`. Tabela de coliziuni dă o altă intrare $i$ în tabela de nume unde se poate găsi cheia căutată. Dacă nu se găseşte valoarea $x$ în această nouă intrare $i$, se continuă cu intrarea $i'$ dată de $i' = col[i]$. Se continuă astfel cât timp $col[i] \neq -1$.

```
static int cauta(String x) {
  for (int i = h(x); i != 1; i = col[i])
    if (x.equals(nume[i]))
      return telefon[i];
  return 1;
}
```

Astfel, procedura de căutare consumă un timp mai mare sau egal ca lungimea medie a claselor de echivalenţă definite pe tabelă de valorile $h(x)$, adică de lungimea medie a listei de coliziuni. Dacă funcţia de dispersie este perfect uniformă, nu apar coliziuni şi determină toate elementele printr-o singură comparaţie. Acest caz este foarte puţin probabil. Dacă numărul mediu de elemente având aceeaşi valoare de dispersie este $k = N/M$, unde $M$ este numărul claselor de echivalenţă definite de $h$, căutarea ia un timp de $N/M$. Dispersia nu face decât să reducă printr-un factor constant timpul căutării secvenţiale. Interesul faţă de dispersie este pentru că adesea este foarte eficace, şi uşor de programat.

# Capitolul 10

# Algoritmi elementari de sortare

Tablourile sunt structuri de bază în informatică. Un tablou reprezintă, în funcţie de dimensiunile sale, un vector sau o matrice cu elemente de acelaşi tip. Un tablou permite accesul direct la un element, şi noi vom utiliza intens această proprietate în algoritmii de sortare pe care îi vom considera.

## 10.1 Introducere

Ce este sortarea? Presupunem că se dă un şir de $N$ numere întregi $a_i$, şi se doreşte aranjarea lor în ordine crescătoare, în sens larg. De exemplu, pentru $N = 10$, şirul

$$18, 3, 10, 25, 9, 3, 11, 13, 23, 8$$

va deveni

$$3, 3, 8, 9, 10, 11, 13, 18, 23, 25.$$

Această problemă este clasică în informatică şi a fost studiată în detaliu, de exemplu, în [23]. În proctică se întâlneşte adesea această problemă. De exemplu, stabilirea clasamentului între studenţi, construirea unui dicţionar, etc. Trebuie făcută o distinţie între sortarea unui număr mare de elemente şi a unui număr mic de elemente. În acest al doilea caz, metoda de sortare este puţin importantă.

Un *algoritm amuzant* constă în a vedea dacă setul de cărţi de joc din mână este deja ordonat. Dacă nu este, *se dă cu ele de pământ* şi se reîncepe. După un anumit timp, *există riscul* de a avea cărţile ordonate. Desigur, poate să nu se termine niciodată, pentru noi, *această sortare*.

O altă tehnică (discutând serios de data aceasta), frecvent utilizată la un joc de cărţi, constă în a vedea dacă există o *transpoziţie* de efectuat. Dacă există, se face interschimbarea cărţilor de joc şi se caută o altă transpoziţie. Procedeul se repetă până când nu mai există transpoziţii. Această metodă funcţionează foarte bine pentru o bună distribuţie a cărţilor.

Este uşor de intuit că numărul obiectelor de sortat este important. Nu este cazul să se caute o metodă sofisticată pentru a sorta 10 elemente. Cu alte cuvinte, *nu se trage cu tunul într-o muscă*!

Exemplele tratate într-un curs sunt întotdeauna de dimensiuni limitate, din considerente pedagogice nu este posibil de a prezenta o sortare a mii de elemente.

## 10.2 Sortare prin selecţie

În cele ce urmează, presupunem că trebuie să sortăm un număr de întregi care se găsesc într-un tablou (vector) $a$. Algoritmul de sortare cel mai simplu este *prin selecţie*. El constă în găsirea poziţiei în tablou a elementului cu valoarea cea mai mică, adică întregul $m$ pentru care $a_i \geq a_m$ pentru orice $i$. Odată găsită această poziţie $m$, se schimbă între ele elementele $a_1$ şi $a_m$.

Apoi se reîncepe această operaţie pentru şirul $(a_2, a_3, ..., a_N)$, tot la fel, căutându-se elementul cel mai mic din acest şir şi interschimbându-l cu $a_2$. Şi aşa mai departe până la un moment dat când şirul va conţine un singur element.

Căutarea celui mai mic element într-un tablou este un prim exerciţiu de programare. Determinarea poziţiei acestui element este foarte simplă, ea se poate efectua cu ajutorul instrucţiunilor următoare:

```
m = 0;
for (int j = 1; j < N; ++j)
  if (a[j] < a[m])
    m = i;
```

Schimbarea între ele a celor două elemente necesită o variabilă temporară $t$ şi se efectuează prin:

```
t = a[m]; a[m] = a[1]; a[1] = t;
```

Acest set de operaţii trebuie reluat prin înlocuirea lui 1 cu 2, apoi cu 3 şi aşa mai departe până la $N$. Aceasta se realizează prin introducerea unei variabile $i$ care ia toate valorile între 1 şi $N$.

Toate acestea sunt arătate în programul care urmează.

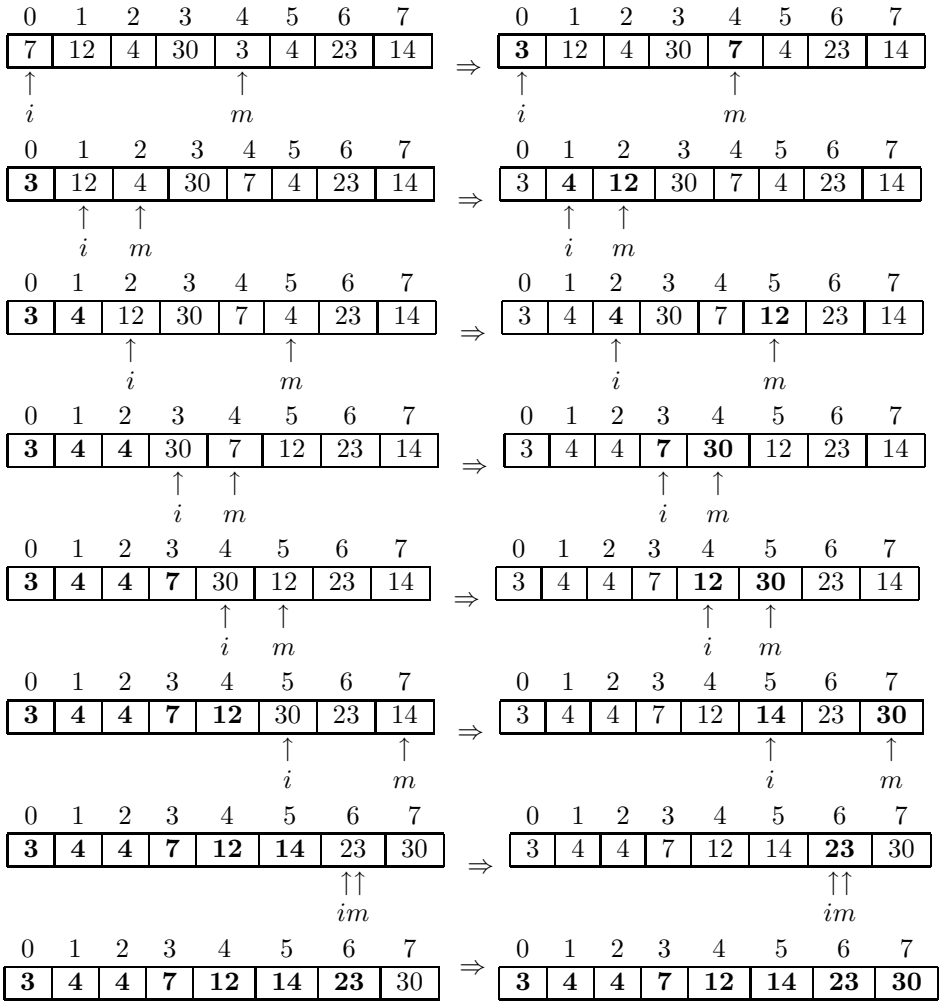
De această dată vom prezenta programul complet.

Procedurile de achiziţionare a datelor şi de returnare a rezultatelor sunt deasemenea prezentate.

Pentru alţi algoritmi, ne vom limita la descrierea efectivă a sortării.

```
class SortSelectie
{
  final static int N = 10;
  static int[] a = new int[N];

  static void initializare()
  {
    int i;
    for (i = 0; i < N; ++i)
      a[i] = (int) (Math.random() * 128);
  }

  static void afisare()
  {
    int i;
    for (i = 0; i < N; ++i)
      System.out.print (a[i] + " ");
    System.out.println();
  }

  static void sortSelectie()
  {
    int min, t;
    int i, j;
    for (i = 0; i < N  1; ++i)
    {
      min = i;
      for (j = i+1; j < N; ++j)
        if (a[j] < a[min])
          min = j;
      t = a[min];
      a[min] = a[i];
      a[i] = t;
    }
  }

  public static void main (String args[])
  {
    initializare();
    afisare();
    sortSelectie();
    afisare();
  }
}
```

Un exemplu de execuţii este:



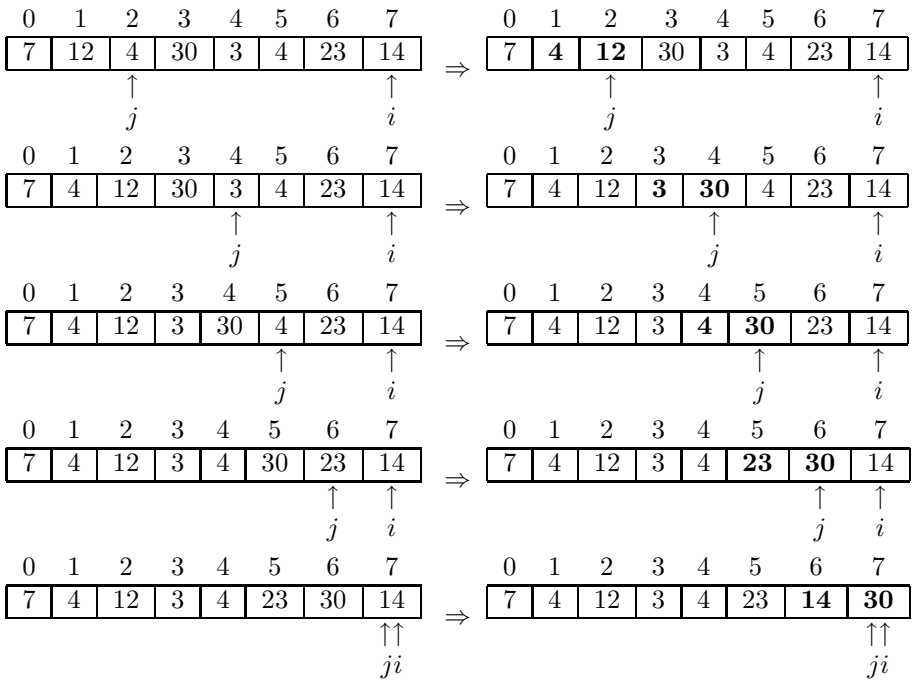

Este uşor de calculat numărul de operaţii necesare. La fiecare iteraţie, se pleacă de la elementul $a_i$ şi se compară succesiv cu $a_{i+1}$, $a_{i+2}$, ..., $a_N$. Se fac deci $N - i$ comparaţii. Se începe cu $i = 1$ şi se termină cu $i = N - 1$. Deci, se fac $(N-1) + (N-2) + ... + 2 + 1 = N(N-1)/2$ comparaţii şi $N - 1$ interschimbări. Sortarea prin selecţie execută un număr de comparaţii de ordinul $N^2$.

O variantă a sortării prin selecţie este **metoda bulelor**. Principiul ei este de a parcurge şirul $(a_1, a_2, ..., a_N)$ inversând toate perechile de elemente consecutive $(a_j - 1, a_j)$ neordonate. După prima parcurgere, elementul maxim se va afla pe poziţia $N$. Se reîncepe cu prefixul $(a_,a_2, ..., a_{N-1})$, ..., $(a_1, a_2)$.

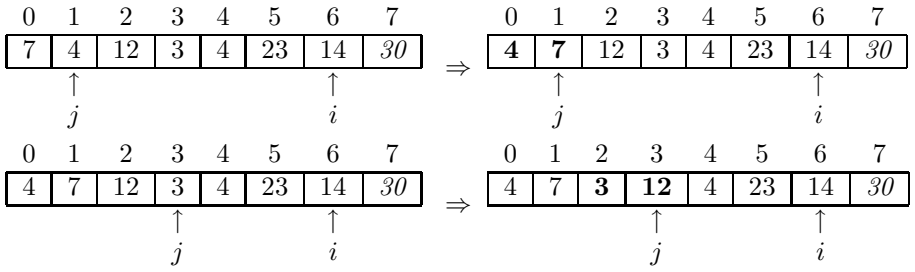
Procedura corespunzătoare utilizează un indice $i$ care marchează sfârşitul prefixului în sortare, şi un indice $j$ care permite deplasarea către marginea $i$.

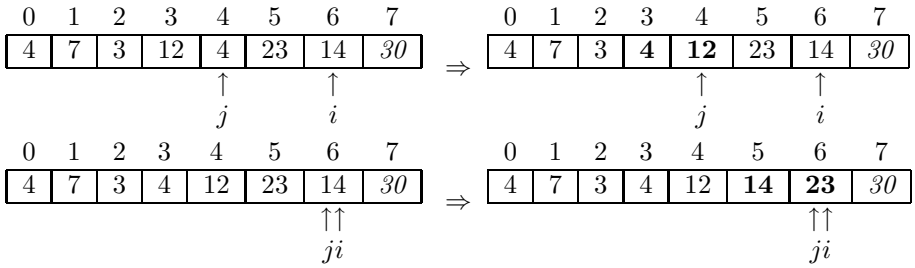
Se poate calcula de asemenea foarte uşor numărul de operaţii şi se obţine un număr de ordinul $O(N^2)$ comparaţii şi, eventual interschimbări (dacă, de exemplu, tabloul este iniţial în ordine descrescătoare).

```
static void sortBule() {
  int t;
  for (int i = N1; i >= 0; i)
    for (int j = 1; j <= i; ++j)
      if (a[j1] > a[j]) {
        t = a[j1]; a[j1] = a[j]; a[j] = t;
      }
}
```

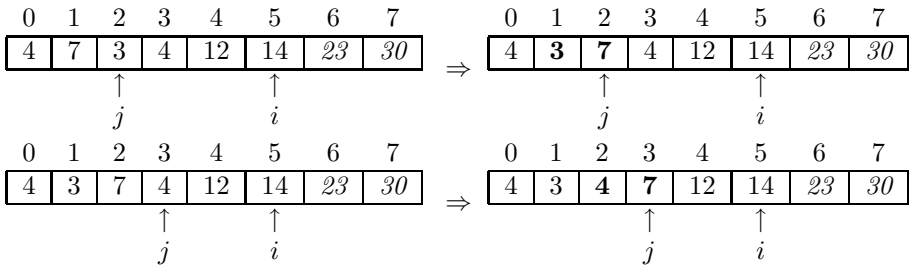



După prima parcurgere **30** a ajuns pe locul său (ultimul loc în vector).

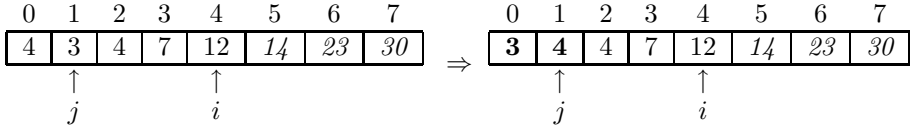





După a doua parcurgere **23** a ajuns pe locul său (penultimul loc în vector).




După a treia parcurgere **14** a ajuns pe locul său (de fapt era deja pe locul său de la începutul acestei parcurgeri; s-au mai *aranjat* totuşi câteva elemente!).




După a patra parcurgere **12** a ajuns pe locul său (de fapt era deja pe locul său de la începutul acestei parcurgeri; oricum, la această parcurgere s-au mai *aranjat* câteva elemente!).

La următoarea parcurgere nu se efectuează nici o interschimbare de elemente. Vectorul este deja sortat, aşa că următoarele parcurgeri se fac, de asemenea, fără să se execute nici o interschimbare de elemente. O idee bună este introducerea unei variabile care să contorizeze numărul de interschimbări din cadrul unei parcurgeri. Dacă nu s-a efectuat nici o interschimbare atunci vectorul este deja sortat aşa că se poate întrerupe execuţia următoarelor parcurgeri. Programul modificat este:

```
static void sortBule() {
  int t, k;
  for (int i = N1; i >= 0; i)
  {
    k=0;
    for (int j = 1; j <= i; ++j)
      if (a[j1] > a[j]) {t = a[j1]; a[j1] = a[j]; a[j] = t; k++;}
    if(k==0) break;
  }
}
```

## 10.3 Sortare prin inserţie

O metodă complet diferită este *sortarea prin inserţie*. Aceasta este metoda utilizată pentru sortarea unui pachet de cărţi de joc. Se ia prima carte, apoi a doua şi se aranjează în ordine crescătoare. Se ia a treia carte şi se plasează pe poziţia corespunzătoare faţă de primele două cărţi, şi aşa mai departe. Pentru cazul general, să presupunem că primele $i-1$ cărţi sun deja sortate crescător. Se ia a $i$-a carte şi se plasează pe poziţia corespunzătoare relativ la primele $i-1$ cărţi deja sortate. Se continuă până la $i = N$.

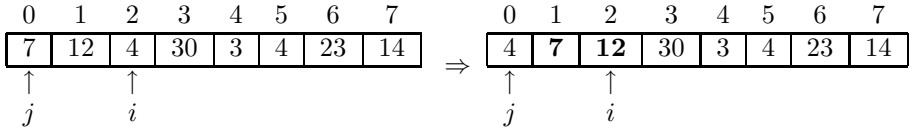
### 10.3.1 Inserţie directă

Dacă determinarea poziţiei corespunzătoare, în subşirul format de primele $i-1$ elemente, se face secvenţial, atunci sortarea se numeşte *sortare prin inserţie directă*. Procedura care realizează aceastsortare este:

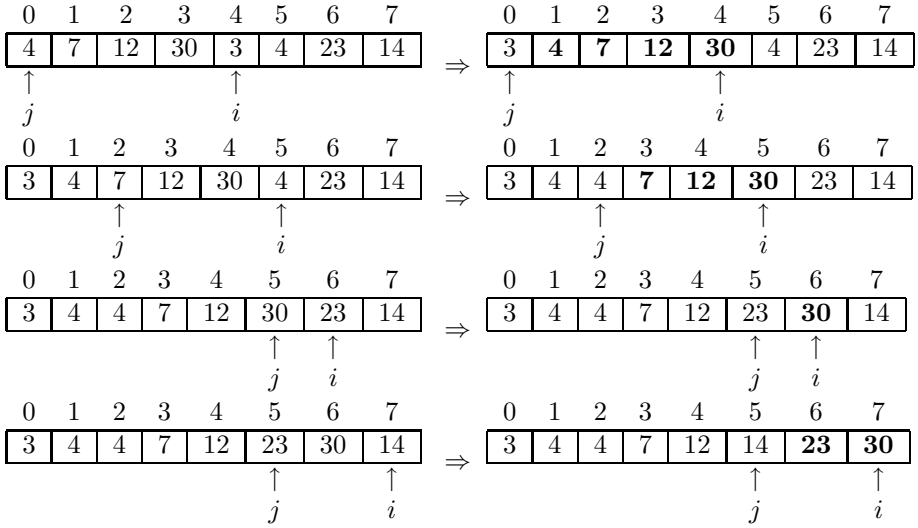
```
static void sortInsertie() {
  int j, v;
  for (int i = 1; i < N; ++i) {
    v = a[i]; j = i;
    while (j > 0 && a[j-1] > v) {
      a[j] = a[j-1];
      j;
    }
    a[j] = v;
  }
}
```

Pentru plasarea elementului $a_i$ din vectorul nesortat (elementele de pe poziţiile din stânga fiind deja sortate) se parcurge vectorul spre stânga plecând de la poziţia $i-1$. Elementele vizitate se deplasează cu o poziţie spre dreapta pentru a permite plasarea elementului $a_i$ (a cărui valoare a fost salvată în variabila temporară $v$) pe poziţia corespunzătoare. Procedura conţine o mică eroare, dacă $a_i$ este cel mai mic element din tablou, căci va ieşi din tablou spre stânga. Se poate remedia această situaţie plasând un element $a_0$ de valoare $-max\_int$. Se spune că a fost plasată o *santinelă* la stânga tabloului $a$. Aceasta nu este întotdeauna posibil, şi atunci trebuie adăugat un test asupra indicelui $j$ în bucla `while`. Un exemplu numeric este cel care urmează:

Inserarea lui **12** nu necesită deplasări de elemente.



Inserarea lui **30** nu necesită deplasări de elemente.




Numărul de comparaţii pentru inserarea unui element în secvenţa deja sortată este egal cu numărul de inversiuni plus 1.

Fie $c_i$ numărul de comparaţii. Atunci

$$c_i = 1 + \text{card}\{a_j | a_j > a_i, j < i\}$$

Pentru o permutare $\pi$ corespunzătoare unui şir de sortări, unde numărul de inversiuni este $inv(\pi)$, numărul total de comparaţii pentru sortarea prin inserţie este

$$C_\pi = \sum_{i=2}^{N} c_i = N - 1 + inv(\pi).$$

Deci, numărul mediu de comparaţii este

$$C_N = \frac{1}{N!} \sum_{\pi \in S_N} C_\pi = N - 1 + \frac{N(N-1)}{4} = \frac{N(N+3)}{4} - 1$$

unde $S_n$ reprezintă grupul permutărilor de $n$ elemente.

Deşi ordinul de creştere este tot $N^2$, această metodă de sortare este mai eficientă decât sortarea prin selecţie. Cea mai bună metodă de sortare necesită $n \log_2 n$ comparaţii.

În plus, ea are o proprietate foarte bună: numărul de operaţii depinde puternic de ordinea iniţială din tablou. În cazul în care tabloul este aproape ordonat, şi drept urmare există puţine inversiuni şi sunt necesare puţine operaţii, spre deosebire de primele două metode de sortare.

Sortarea prin inserare este deci o metodă de sortare bună dacă tabloul de sortat are şansa de a fi *aproape ordonat*.

### 10.3.2 Inserţie binară

Metoda anterioară se poate îmbunatăţi dacă ţinem cont de faptul că secvenţa în care se face inserarea este deja ordonată, iar în loc să se facă inserţia directă în această secvenţă, căutarea poziţiei pe care se face inserarea se face prin căutare binară. Un program complet este:

```
import java.io.*;
class SortInsBinApl
{
  static int[] a={3,8,5,4,9,1,6,4};

  static void afiseaza()
  {
    int j;
    for(j=0; j<a.length; j++)
      System.out.print(a[j] + " ");
    System.out.println();
  }

  static int pozitiaCautBin(int p, int u, int x)
  {
    int i=u+1;
    while (p <= u)
    {
      i=(p+u)/2;
      if (x>a[i])
        p=i+1;
        else if (x<a[i])
               u=i-1;
               else return i;
    }
    return p;
  }

  static void deplasare(int k, int i)
  {
    if (i != k)
    {
      int x=a[k];
      for(int j=k; j>=i+1; j--) a[j]=a[j-1];
      a[i]=x;
    }
  }
```

```
  static void sorteaza()
  {
    int N=a.length,i;
    for(int k=1;k<=N-1;k++)
    {
      i=pozitiaCautBin(0,k-1,a[k]);
      deplasare(k,i);
    }
  }

  public static void main(String[] args)
  {
    afiseaza();
    sorteaza();
    afiseaza();
  }
}
```

## 10.4 Sortare prin interschimbare

Această metodă foloseşte interschimbarea ca şi caracteristică principală a metodei de sortare. În cadrul acestei metode se compară şi se interschimbă perechi adiacente de chei până când toate elementele sunt sortate.

```
static void interschimbare(int a[])
{
  int x, i, n=a.length;
  boolean schimb = true;
  while (!schimb)
  {
    schimb=false;
    for(i=0; i<n-1; i++)
      if (a[i]>a[i+1])
      {
        x = a[i];
        a[i] = a[i+1];
        a[i+1] = x;
        schimb=true;
      }
  }
}
```

## 10.5 Sortare prin micşorarea incrementului - shell

Prezentăm metoda pe următorul şir:

$$44, 55, 12, 42, 94, 18, 6, 67.$$

Se grupează elementele aflate la 4 poziţii distanţă, şi se sortează separat. Acest proces este numit numit *4 sort*. Rezultă şirul:

$$44, 18, 06, 42, 94, 55, 12, 67$$

Apoi se sortează elementele aflate la 2 poziţii distantă. Rezultă:

$$6, 18, 12, 42, 44, 55, 94, 97$$

Apoi se sortează sirul rezultat într-o singură trecere: 1 - sort

$$6, 12, 18, 42, 44, 55, 94, 97$$

Se observă următoarele:

- un proces de sortare $i - sort$ combină 2 grupuri sortate în procesul $2i - sort$ anterior
- în exemplul anterior s-a folosit secventa de incrementi 4, 2, 1 dar orice secvenţă, cu condiţia ca cea mai fină sortare să fie $1 - sort$. În cazul cel mai defavorabil, în ultimul pas se face totul, dar cu multe comparaţii şi interschimbări.
- dacă cei t incrementi sunt $h_1$, $h_2$, .. $h_t$, $h_t = 1$ şi $h_{i+1} < h_i$, fiecare $h_i$-sort se poate implementa ca şi o sortate prin insertie directă.

```
void shell(int a[], int n)
{
  static int h[] = {9, 5, 3, 1};
  int m, x, i, j, k, n=a.length;
  for (m=0; m<4; m++)
  {
    k = h[m];
    /* sortare elemente aflate la distanta k in tablul a[] */
    for (i=k; i<n; i++)
    {
      x = a[i];
      for (j = i-k; (j>=0) && (a[j]>x); j-=k) a[j+k] = a[j];
      a[j+k] = x;
    }
  }
}
```

## 10.6 Sortare prin partitionare - quicksort

Se bazează pe metoda interschimbării, însă din nou, interschimbarea se face pe distanţe mai mari. Astfel, având tabloul $a[]$, se aplică următorul algoritm:

1. se alege la întâmplare un element $x$ al tabloului
2. se scanează tabloul $a[]$ la stânga lui $x$ până când se găseşte un element $a_i > x$
3. se scanează tabloul la dreapta lui $x$ până când se găseşte un element $a_j < x$
4. se interschimbă $a_i$ cu $a_j$
5. se repetă paşii 2, 3, 4 până când scanările se vor întâlni pe undeva la mijlocul tabloului. În acel moment, tabloul $a[]$ va fi partitionat în 2 astfel, la stânga lui $x$ se vor găsi elemente mai mici ca şi $x$, la dreapta, elemente mai mari ca şi $x$. După aceasta, se aplică acelaşi proces subşirurilor de la stânga şi de la dreapta lui $x$, până când aceste subşiruri sunt suficient de mici (se reduc la un singur element).

```
void quicksort(int a[])
{
  int n=a.length;
  int l=0, r=n-1;
  int i=l, j=r;
  int x, temp;
  if (l<r)
  {
    x = a[(l+r)/2];
    do
    {
      while (a[i]<x) i++;
      while (a[j]>x) --j;
      if (i<=j)
      {
        temp=a[i]; a[i]=a[j]; a[j]=temp;
        j--;
        i++;
      }
    } while (i<=j);
    if (l<j) quicksort(a+l, j-l);
    if (i<r) quicksort(a+i, r-i);
  }
}
```

Aceasta metoda are complexitatea $n \log n$, în practică (în medie)!

# Capitolul 11

# Liste

Scopul listelor este de a genera un ansamblu finit de elemente al cărui număr nu este fixat apriori. Elementele acestui ansamblu pot fi numere întregi sau reale, şiruri de caractere, obiecte informatice complexe, etc. Nu suntem acum interesaţi de elementele acestui ansamblu ci de operaţiile care se efectuează asupra acestuia, independent de natura elementelor sale.

Listele sunt obiecte dinamice, în sensul că numărul elementelor variază în cursul execuţiei programului, prin adăugări sau ştergeri de elemente pe parcursul prelucrării. Mai precis, operaţiile permise sunt:

- testarea dacă ansamblul este vid
- adăugarea de elemente
- verificarea dacă un element este în ansamblu
- ştergerea unui element

## 11.1 Liste liniare

Fiecare element al listei este conţinut într-o *celulă* care conţine în plus adresa elementului următor, numit şi *pointer*. Java permite realizarea listelor cu ajutorul claselor şi obiectelor: celulele sunt obiecte (adică instanţe ale unei clase) în care un câmp conţine o referinţă către celula următoare. Referinţa către prima celulă este conţinută într-o variabilă.

```
class Lista {
  int continut;
  Lista urmator;
```

```
  Lista (int x, Lista a) {
    continut = x;
    urmator = a;
  }
}
```

Instrucţiunea `new Lista(x,a)` construieşte o nouă celulă cu câmpurile $x$ şi $a$. Func tia `Lista(x,a)` este un constructor al clasei *Lista* (un constructor este o funcţie nestatică care se distinge prin tipul rezultatului său care este cel al clasei curente, şi prin absenţa numelui de identificare). Obiectul `null` aparţine tuturor claselor şi reprezintă în cazul listelor marcajul de sfârşit de listă. De asemenea `new Lista(2, new Lista(7, new Lista(11,null)))` reprezintă lista $2, 7, 11$.

```
static boolean esteVida (Lista a) {
  return a == null;
}
```

Procedura `adauga` inserează un element în capul listei. Această modalitate de a introduce elemente în capul listei este utilă pentru ca numărul de operaţii necesare adăugării de elemente să fie independent de mărimea listei; este suficientă modificarea valorii capului listei, ceea ce se face simplu prin:

```
static Lista adaug (int x, Lista a) {
  return new Liste (x, a); // capul vechi se va regasi dupa x
}
```

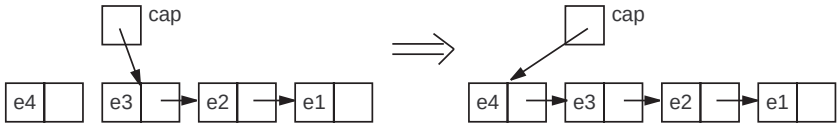



Figura 11.1: Adăugarea unui element în listă

Funcţia `cauta`, care verifică dacă elementul $x$ este în lista $a$, efectuează o parcurgere a listei. Variabila $a$ este modificată iterativ prin `a=a.urmator` pentru a parcurge elementele listei până se găseşte $x$ sau până se găseşte sfârşitul listei ($a = null$).

```
static boolean cauta (int x, Lista a) {
  while (a != null) {
    if (a.continut == x)
      return true;
    a = a.urmator;
  }
  return false;
}
```

Funcţia `cauta` poate fi scrisă în mod recursiv:

```
static boolean cauta (int x, Lista a) {
  if (a == null)
    return false;
  else if (a.continut == x)
         return true;
       else
         return cauta(x, a.urmator);
}
```

sau sub forma:

```
static boolean cauta (int x, Lista a) {
  if (a == null)
    return false;
  else return (a.continut == x) || cauta(x, a.urmator);
}
```

sau sub forma:

```
static boolean cauta (int x, Lista a) {
  return a != null && (a.continut == x || cauta(x, a.urmator));
}
```

Această sciere recursivă este sistematică pentru funcţiile care operează asupra listelor. Tipul *listă* verifică ecuaţia următoare:

$$Lista = \{Lista\_vida\} \oplus Element \times Lista$$

unde $\oplus$ este *sau exclusiv* iar $\times$ este produsul cartezian. Toate procedurile sau funcţiile care operează asupra listelor se pot scrie recursiv în această manieră. De exemplu, lungimea listei se poate determina prin:

```
static int lungime(Lista a) {
  if (a == null) return 0;
  else return 1 + longime(a.urmator);
}
```

care este mai elegantă decât scrierea iterativă:

```
static int lungime(Lista a) {
  int lg = 0;
  while (a != null) {
    ++lg;
    a = a.urmator;
  }
  return lg;
}
```

Alegerea între maniera recursivă sau iterativă este o problemă subiectivă în general. Pe de altă parte se spune că scrierea iterativă este mai eficientă pentru că foloseşte mai puţină memorie. Graţie noilor tehnici de compilare, acest lucru este mai puţin adevărat; ocuparea de memorie suplimentară, pentru calculatoarele de astăzi, este o problemă foarte puţin critică.

Eliminarea unei celule care conţine $x$ se face modificând valoarea câmpului *urmator* conţinut în predecesorul lui $x$: succesorul predecesorului lui $x$ devine succesorul lui $x$. Un tratament particular trebuie făcut dacă elementul care trebuie eliminat este primul element din listă. Procedura recursivă de eliminare este foarte compactă în definiţia sa:

```
static Lista elimina (int x, Lista a) {
  if (a != null)
    if (a.continut == x)
      a = a.urmator;
    else
      a.urmator = elimina (x, a.urmator);
  return a;
}
```

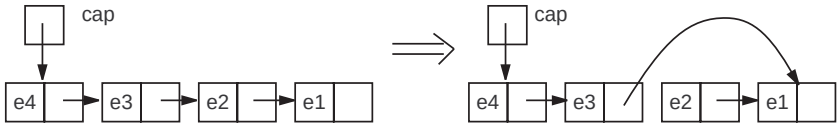



Figura 11.2: Eliminarea unui element din listă

O procedură iterativă solicită un plus de atenţie.

```
static Lista elimina (int x, Lista a) {
  if (a != null)
    if (a.continut == x)
      a = a.urmator;
    else {
      Lista b = a ;
      while (b.urmator != null && b.urmator.continut != x)
        b = b.urmator;
      if (b.urmator != null)
        b.urmator = b.urmator.urmator;
    }
  return a;
}
```

În cadrul funcţiilor anterioare, care modifică lista $a$, nu se dispune de două liste distincte, una care conţine $x$ şi alta identică cu precedenta dar care nu mai conţine $x$. Pentru a face acest lucru, trebuie recopiată o parte a listei $a$ într-o nouă listă, cum face programul următor, unde există o utilizare puţin importantă de memorie suplimentară. Oricum, spaţiul de memorie pierdut este recuperat de culegătorul de spaţiu de memorie GC (Garbage Colection) din Java, dacă nu mai este folosită lista $a$. Cu tehnicile actuale ale noilor versiuni ale GC, recuperarea se efectuează foarte rapid. Aceasta este o diferenţă importantă faţă de limbajele de programare precum Pascal sau C, unde trebuie să fim preocupaţi de spaţiul de memorie pierdut, dacă trebuie să- l reutilizăm.

```
static Lista elimina (int x, Lista a) {
  if (a != null)
    return null;
  else if (a.continut == x)
         return a.urmator;
       else
         return new Lista (a.continut, elimina (x, a.urmator));
}
```

Există o tehnică, utilizată adesea, care permite evitarea unor teste pentru eliminarea unui element dintr-o listă şi, în general, pentru simplificarea programării operaţiilor asupra listelor. Aceasta constă în utilizarea unui *fanion / santinelă* care permite tratarea omogenă a listelor indiferent dacă sunt vide sau nu. În reprezentarea anterioară lista vidă nu a avut aceeaşi structură ca listele nevide. Se utilizează o celulă plasată la începutul listei, care nu are informaţie semnificativă în câmpul *continut*; adresa adevăratei prime celule se află în câmpul *urmator* al acestei celule. Astfel obţinem o listă *păzită*, avantajul fiind că lista vidă conţine numai *garda* şi prin urmare un număr de programe, care făceau un caz special din lista vidă sau din primul element din listă, devin mai simple. Această noţiune este un pic echivalentă cu noţiunea de *santinelă* pentru tablouri. Se utilizează această tehnică în proceduri asupra şirurilor prea lungi.

De asemenea, se pot defini liste circulare cu *gardă / paznic* în care în prima celulă în câmpul *urmator* este ultima celulă din listă.

Pentru implementarea listelor se pot folosi perechi de tablouri : primul tablou, numit *continut*, conţine elementele de informaţii, iar al doilea, numit *urmator*, conţine adresa elementului următor din tabloul *continut*.

Procedura *adauga* efectuează numai 3 operaţii elementare. Este deci foarte eficientă. Procedurile *cauta* şi *elimina* sunt mai lungi pentru că trebuie să parcurgă întreaga listă. Se poate estima că numărul mediu de operaţii este jumătate din lungimea listei. Căutarea binară efectuează un număr logaritmic iar căutarea cu tabele *hash* (de dispersie) este şi mai rapidă.

**Exemplu 1**. Considerăm construirea unei liste de numere prime mai mici decât un număr natural $n$ dat. Pentru construirea acestei liste vom începe, în

prima fază, prin adăugarea numerelor de la 2 la $n$ începând cu $n$ şi terminând cu 2. Astfel numerele vor fi în listă în ordine crescătoare. Vom utiliza metoda clasică a ciurului lui Eratostene: considerăm succesiv elementele listei în ordine crescătoare şi suprimăm toţi multiplii lor. Aceasta se realizează prin procedura următoare:

```
static Lista Eratostene (int n) {
  Lista a=null;
  int k;
  for(int i=n; i>=2; --i) {
    a=adauga(i,a);
  }
  k=a.continut;
  for(Lista b=a; k*k<=n; b=b.urmator) {
    k=b.continut;
    for(int j=k; j<=n/k; ++j)
      a=elimina(j*k,a);
  }
  return a;
}
```

**Exemplu 2**. Pentru fiecare intrare $i$ din intervalul $[0..n-1]$ se construieşte o listă simplu înlănţuită formată din toate cheile care au $h(x) = i$. Elementele listei sunt intrări care permit accesul la tabloul de nume şi numere de telefon. Procedurile de căutare şi inserare a unui element $x$ într-un tablou devin proceduri de căutare şi adăugare într-o listă. Tot astfel, dacă funcţia $h$ este rău aleasă, numărul de coliziuni devine important, multe liste devin de dimensiuni enorme şi fărâmiţarea riscă să devină la fel de costisitoare ca şi căutarea într-o listă simplu înlănţuită obişnuită. Dacă $h$ este bine aleasă, căutarea este rapidă.

```
Lista al[] = new Lista[N1];

static void insereaza (String x, int val) {
  int i = h(x);
  al[i] = adauga (x, al[i]);
}

static int cauta (String x) {
  for (int a = al[h(x)]; a != null; a = a.urmator) {
    if (x.equals(nume[a.continut]))
      return telefon[a.continut];
  }
  return -1;
}
```

## 11.2 Cozi

Cozile sunt liste liniare particulare utilizate în programare pentru gestionarea obiectelor care sunt în aşteptarea unei prelucrări ulerioare, de exemplu procesele de aşteptare a resurselor unui sistem, nodurile unui graf, etc. Elementele sunt adăugate sistematic la coadă şi sunt eliminate din capul listei. În engleză se spune strategie FIFO (First In First Out), în opoziţie cu strategia LIFO (Last In Firs Out) utilizată la stive.

Mai formalizat, considerăm o mulţime de elemente $E$, mulţimea cozilor cu elemente din $E$ este notată $Coada(E)$, coada vidă (care nu conţine nici un element) este $C_0$, iar operaţiile asupra cozilor sunt: $esteVida$, $adauga$, $valoare$, $elimina$:

- $esteVida$ este o aplicaţie definită pe $Coada(E)$ cu valori în $\{true, false\}$, $esteVida(C)$ aste egală cu $true$ dacă şi numai dacă coada $C$ este vidă.
- $adauga$ este o aplicaţie definită pe $E \times Coada(E)$ cu valori în $Coada(E)$, $adauga(x, C)$ este coada obţinută plecând de la coada $C$ şi inserând elementul $x$ la sfârşitul ei.
- $valoare$ este o aplicaţie definită pe $Coada(E) - C_0$ cu valori în $E$ care asociază unei cozi $C$, nevidă, elementul aflat în *cap*.
- $elimina$ este o aplicaţie definită pe $Coada(E) - C_0$ cu valori în $Coada(E)$ care asociază unei cozi nevide $C$ o coadă obţinută plecând de la $C$ şi eliminând primul element.

Operaţiile asupra cozilor satisfac următoarele relaţii:

- Pentru $C \neq C_0$
  - $elimina(adauga(x, C)) = adauga(x, elimina(C))$
  - $valoare(adauga(x, C)) = valoare(C)$
- Pentru toate cozile $C$
  - $esteVida(adauga(x, C)) = false$
- Pentru coada $C_0$
  - $elimina(adauga(x, C_0)) = C_0$
  - $valoare(adauga(x, C_0)) = x$
  - $esteVida(C_0) = true$

O primă idee de realizare sub formă de programe a operaţiilor asupra cozilor este de a împrumuta tehnica folosită în locurile unde clienţii stau la coadă pentru a fi *serviţi*, de exemplu la gară pentru a lua bilete, sau la casă într-un magazin.

Fiecare client care se prezintă obţine un număr şi clienţii sunt apoi chemaţi de către funcţionarul de la ghişeu în ordinea crescătoare a numerelor de ordine primite la sosire. Pentru gestionarea acestui sistem, gestionarul trebuie să cunoască două numere: numărul obţinut de către ultimul client sosit şi numărul obţinut de către ultimul client servit. Notăm aceste două numere *inceput* şi $sfarsit$ şi gestionăm sistemul în modul următor:

- coada de aşteptare este vidă dacă şi numai dacă $inceput = sfarsit$,
- atunci când soseşte un nou client, se incrementează $sfarsit$ şi se dă acest număr clientului respectiv,
- atunci când funcţionarul este liber el poate servi un alt client, dacă coada nu este vidă, incrementează *inceput* şi cheamă posesorul acestui număr.

În cele ce urmează sunt prezentate toate operaţiile în Java utilizând tehnicile următoare: se reatribuie numărul 0 unui nou client atunci când se depăşeşte un anumit prag pentru valoarea $sfarsit$. Se spune că este un tablou (sau tampon) circular, sau coadă circulară.

```
class Coada {
  final static int MaxC = 100;
  int inceput;
  int sfarsit;
  boolean plina, vida;
  int continut[];

  Coada () {
    inceput = 0; sfarsit = 0;
    plina= false; vida = true;
    info = new int[MaxC];
  }

  static Coada vida(){
    return new Coada();
  }

  static void facVida (Coada c) {
    c.inceput = 0; c.sfarsit = 0;
    c.plina = false; c.vida = true;
  }

  static boolean esteVida(Coada c) {
    return c.vida;
  }
```

```
  static boolean estePlina(Coada c) {
    return c.plina;
  }

  static int valoare(Coada c) {
    if (c.vida)
      erreur ("Coada Vida.");
    return c.info[f.inceput];
  }

  private static int succesor(int i) {
    return (i+1) % MaxC;
  }

  static void adaug(int x, Coada c) {
    if (c.plina)
      erreur ("Coada Plina.");
    c.info[c.sfarsit] = x;
    c.sfarsit = succesor(c.sfarsit);
    c.vida = false;
    c.plina = c.sfarsit == c.inceput;
  }

  static void elimina (Coada c) {
    if (c.vida)
      erreur ("Coada Vida.");
    c.inceput = succesor(c.inceput);
    c.vida = c.sfarsit == c.inceput;
    c.plina = false;
  }
}
```

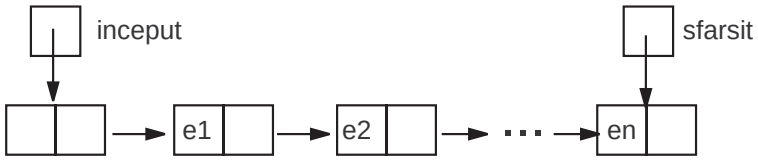



Figura 11.3: Coadă de aşteptare implementată ca listă

O altă modalitate de gestionare a cozilor constă în utilizarea listelor înlănţuite cu

*gardă* în care se cunosc adresele primului şi ultimului element. Aceasta dă operaţiile următoare:

```
class Coada {
  Lista inceput;
  Lista sfarsit;

  Coada (Lista a, Lista b) {
    inceput = a;
    sfarsit = b;
  }

  static Coada vida() {
    Lista garda = new Lista();
    return new Coada (garda, garda);
  }

  static void facVida (Coada c) {
    Lista garda = new Lista();
    c.inceput = c.sfarsit = garda;
  }

  static boolean esteVida (Coada c) {
    return c.inceput == c.sfarsit;
  }

  static int valoare (Coada c) {
    Lista b = c.inceput.next;
    return b.info;
  }

  static void adauga (int x, Coada c) {
    Lista a = new Lista (x, null);
    c.sfarsit.next = a;
    f.sfarsit = a;
  }

  static void elimina (Coada c) {
    if (esteVida(c))
      erreur ("Coada Vida.");
    c.inceput = c.inceput.next;
  }
}
```

Cozile se pot implementa fie cu ajutorul tablourilor fie cu ajutorul listelor sim-

plu înlănţuite. Scrierea programelor constă în primul rând în alegerea structurilor de date pentru reprezentarea cozilor. Ansamblul funcţiilor care operează asupra cozilor se pot plasa într-un *modul* care manipulează tablouri sau liste. Utilizarea cozilor se face cu ajutorul funcţiilor *vida*, *adauga*, *valoare*, *elimina*. Aceasta este deci *interfaţa* cozilor care are importanţă în programe complexe.

## 11.3 Stive

Noţiunea de stivă intervine în mod curent în programare, rolul său principal fiind la implementarea apelurilor de proceduri. O stivă se poate imagina ca o cutie în care sunt plasate obiecte şi din care se scot în ordinea inversă faţa de cum au fost introduse: obiectele sunt puse unul peste altul în cutie şi se poate avea acces numai la obiectul situat în vârful stivei. În mod formalizat, se consideră o mulţime $E$, mulţimea stivelor cu elemente din $E$ este notată $Stiva(E)$, stiva vidă (care nu conţine nici un element) este $S_0$, operaţiile efectuate asupra stivelor sunt *vida*, *adauga*, *valoare*, *elimina*, ca şi la fire. De această dată, relaţiile satisfăcute sunt următoarele:

- $elimina(adauga(x, S)) = S$
- $esteVida(adauga(x, S)) = false$
- $valoare(adauga(x, S)) = x$
- $esteVida(S_0) = true$

Cu ajutorul acestor relaţii se pot exprima toate operaţiile relative la stive.

Realizarea operaţiilor asupra stivelor se poate face utilizând un tablou care conţine elementele şi un indice care indică poziţia vârfului stivei.

```
class Stiva {
  final static int maxP = 100;
  int inaltime;
  Element continut[];

  Stiva() {
    inaltime = 0;
    continut = new Element[maxP];
  }

  static Fir vid () {
    return new Stiva();
  }
```

```
  static void facVida (Stiva s) {
    s.inaltime = 0;
  }

  static boolean esteVida (Stiva s) {
    return s.inaltime == 0;
  }

  static boolean estePlina (Stiva s) {
    return s.inaltime == maxP;
}

  static void adauga (Element x, Stiva s) throws ExceptionStiva {
    if (estePlina (s))
      throw new ExceptionStiva("Stiva plina");
    s.continut[s.inaltime] = x;
    ++ s.inaltime;
  }

  static Element valoare (Stiva s) throws ExceptionStiva {
    if (esteVida (s))
      throw new ExceptionStiva("Stiva vida");
    return s.continut[s.inaltime-1];
  }

  static void supprimer (Stiva s) throws ExceptionStiva {
    if (esteVida (s))
      throw new ExceptionStiva ("Stiva vida");
     s.inaltime;
  }
}
```

unde

```
class ExceptionStiva extends Exception {
  String text;

  public ExceptionStiva (String x) {
    text = x;
  }
}
```

Stivele se pot implementa atât cu ajutorul tablourilor (vectorilor) cât şi cu ajutorul listelor simplu înlănţuite.

## 11.4 Evaluarea expresiilor aritmetice prefixate

Vom ilustra utilizarea stivelor printr-un program de evaluare a expresiilor aritmetice scrise sub o formă particulară. În programare o expresie aritmetică poate conţine numere, variabile şi operaţii aritmetice (ne vom limita numai la + şi *). Vom considera ca intrări numai numere naturale.

Expresiile prefixate conţin simbolurile: numere naturale, +, *, ( şi ). Dacă $e_1$ şi $e_2$ sunt expresii prefixate atunci $(+e_1e_2)$ şi $(*e_1e_2)$ sunt expresii prefixate.

Pentru reprezentarea unei expresii prefixate în Java, vom utiliza un tablou ale cărui elemente sunt entităţile expresiei. Elementele tabloului sunt obiecte cu trei câmpuri: primul reprezintă natura entităţii (simbol sau număr), al doilea reprezintă valoarea entităţii dacă aceasta este număr, iar al treilea este este un simbol dacă entitatea este simbol.

```
class Element {
  boolean esteOperator;
  int valoare;
  char simbol;
}
```

Vom utiliza funcţiile definite pentru stivă şi procedurile definite în cele ce urmează.

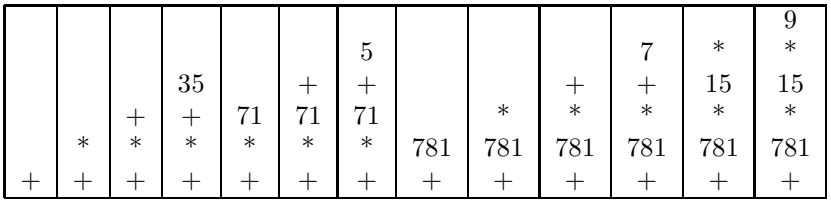
```
static int calcul (char a, int x, int y) {
  switch (a) {
    case '+': return x + y;
    case '*': return x * y;
  }
  return 1;
}
```

Procedura de evaluare constă în stivuirea rezultatelor intermediare, stiva conţinând operatori şi numere, dar niciodată nu va conţine consecutiv numere. Se examinează succesiv entităţile expresiei dacă entitatea este un operator sau un număr şi dacă vârful stivei este un operator, atunci se plasează în stivă. Dacă este un număr şi vârful stivei este de asemenea un număr, acţionează operatorul care precede vârful stivei asupra celor două numere şi se repetă operaţia asupra rezultatului găsit.

De exemplu, pentru expresia

```
(+ (* (+ 35 36) (+ 5 6)) (* (+ 7 8) (*9 9 )))
```

evaluarea decurge astfel:



```
static void insereaza (Element x, Stiva s) throws ExceptionStiva {
  Element y, op;
  while (!(Stiva.esteVida(s) || x.esteOperator
                             || Stiva.valoare(s).esteOperator)) {
    y = Stiva.valoare(s);
    Stiva.elimina(s);
    op = Stiva.valoare(s);
    Stiva.elimina(s);
    x.valoare = calcul(op.valsimb, x.valoare, y.valoare);
  }
  Stiva.adauga(x,s);
}

static int calcul (Element u[]) throws ExceptionStiva {
  Stiva s = new Stiva();
  for (int i = 0; i < u.length ; ++i) {
    insereaza(u[i], s);
  }
  return Stiva.valoare(s).valoare;
}
```

În acest caz, este utilă prezentarea unui program principal care utilizează aceste funcţii.

```
public static void main (String args[]) {
  Element exp[] = new Element [args.length];
  for (int i = 0; i < args.length; ++i) {
    String s = args[i];
    if (s.equals("+") || s.equals("*"))
      exp[i] = new Element (true, 0, s.charAt(0));
    else
      exp[i] = new Element (false, Integer.parseInt(s), ' ');
  }
  try { System.out.println(calcul(exp)); }
  catch (ExceptionStiva x) {
    System.err.println("Stiva " + x.nom);
  }
}
```

## 11.5 Operaţii asupra listelor

În această secţiune sunt prezentaţi câţiva algoritmi de manipulare a listelor. Aceştia sunt utilizaţi în limbajele de programare care au listele ca structuri de bază. Funcţia `Tail` este o primitivă clasică; ea suprimă primul element al listei.

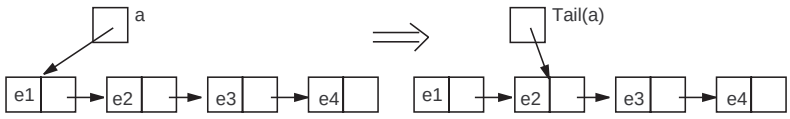



Figura 11.4: Suprimarea primului element din listă

```
class Lista {
  Object info;
  Lista next;
  Lista(Object x, Lista a) {
    info = x;
    next = a;
  }

  static Lista cons (Object x, Lista a) {
    return new Lista (x, a);
  }

  static Object head (Lista a) {
    if (a == null)
      erreur ("Head d'une liste vide.");
    return a.info;
  }

  static Lista tail (Lista a) {
    if (a == null)
      erreur ("Tail d'une liste vide.");
    return a.next;
}
```

Procedurile asupra listelor construiesc o listă plecând de la alte două liste, pun o listă la capătul celeilalte liste. În prima procedură `append`, cele două liste nu sunt modificate; în a doua procedură, `concat`, prima listă este transformată pentru a reţine rezultatul. Totusi, se poate remarca faptul că, dacă `append` copiază primul său argument, partajează finalul listei rezultat cu al doilea argument.

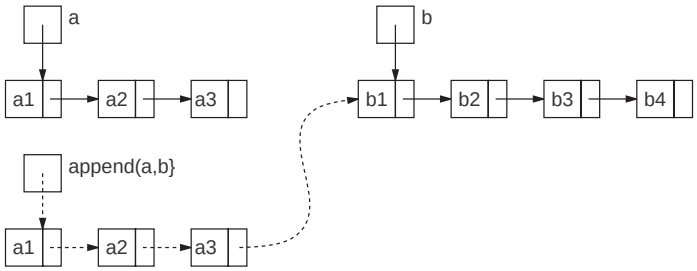



Figura 11.5: Concatenarea a două liste prin *append*

```
static Lista append (Lista a, Lista b) {
  if (a == null)
    return b;
  else
    return adauga(a.info, append (a.next, b)) ;
}
```

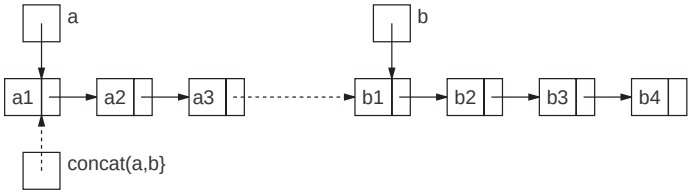



Figura 11.6: Concatenarea a două liste prin *concat*

```
static Lista concat(Lista a, Lista b)
{
  if (a == null)
    return b;
  else
  {
    Lista c = a;
    while (c.next != null)
      c = c.next;
    c.next = b;
    return a;
  }
}
```

Această ultimă procedură se poate scrie recursiv:

```
static Lista concat (Lista a, Lista b) {
  if (a == null)
    return b;
  else {
    a.next = concat (a.next, c);
    return a;
  }
}
```

Procedura de calcul de *oglindire* a unei liste $a$ constă în construirea unei liste în care elementele listei $a$ sunt în ordine inversă. Realizarea acestei proceduri este un exerciţiu clasic de programare asupra listelor. Dăm aici două soluţi, una iterativă, cealaltă recursivă, complexitatea este $O(n^2)$ deci pătratică, dar clasică. Noua metodă `nReverse` modifică argumentul său, în timp ce `Reverse` nu-l modifică ci construieşte o altă listă pentru rezultat.

```
static Lista nReverse (Lista a) {
  Lista b = null;
  while (a != null) {
    Lista c = a.next;
    a.next = b; b = a; a = c;
  }
  return b;
}
```

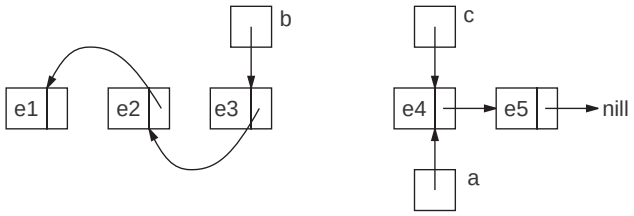



Figura 11.7: Transformarea listei prin inversarea legăturilor

```
static Lista Reverse (Lista a) {
  if (a == null)
    return a;
  else
    return append(Reverse (a.next), adauga(a.info, null));
}
```

Se poate scrie o versiune iterativă a versiunii recursive, graţie unei funcţii auxiliare care acumulează rezultatul într-unul din argumentele sale:

```
static Lista nReverse (Lista a) {
  return nReverse1(null, a);
}

static Lista nReverse1 (Lista b, Lista a) {
  if (a == null)
    return b;
  else
    return nReverse1(adauga(a.info, b), a.next);
}
```

Un alt exerciţiu formator constă în a administra liste în care elementele sunt aranjate în ordine crescătoare. Procedura de adăugare devine ceva mai complexă pentru că trebuie găsită poziţia celulei care trebuie adăugată după parcurgerea unei părţi a listei.

Nu tratăm acest exerciţiu decât în cazul listelor circulare cu gardă. Pentru o astfel de listă, valoarea câmpului $info$ din prima celulă nu are nici o semnificaţie (este *celula de gardă*). Câmpul $next$ din ultima celulă conţine adresa primei celule.

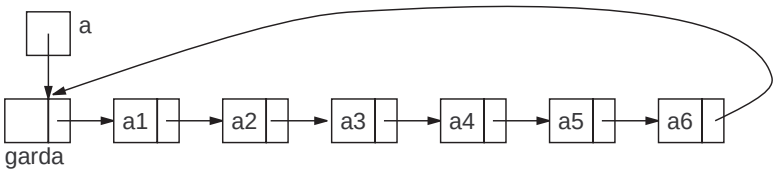



Figura 11.8: Listă circulară cu gardă

```
static Lista insereaza (int v, Lista a) {
  Lista b = a;
  while (b.next != a && v > head(b.next))
    b = b.next;
  b.next = adauga(v, b.next);
  a.info = head(a) + 1;
  return a;
}
```

# Capitolul 12

# Algoritmi divide et impera

## 12.1 Tehnica divide et impera

Divide et impera[1] este o tehnică de elaborare a algoritmilor care constă în:

- Descompunerea repetată a problemei (subproblemei) ce trebuie rezolvată în subprobleme mai mici.
- Rezolvarea în acelaşi mod (recursiv) a tuturor subproblemelor.
- Compunerea subsoluţiilor pentru a obţine soluţia problemei (subproblemei) iniţiale.

Descompunerea problemei (subproblemelor) se face până în momentul în care se obţin subprobleme de dimensiuni atât de mici încât au soluţie cunoscută sau pot fi rezolvate prin tehnici elementare.

Metoda poate fi descrisă astfel:

**procedure** divideEtImpera($P, n, S$)
  **if** ($n \leq n_0$)
    **then** *rezolvă* subproblema $P$ prin tehnici elementare
    **else**
      *împarte* $P$ în $P_1, ..., P_a$ de dimensiuni $n_1, ..., n_a$
      divideEtImpera($P_1, n_1, S_1$)
      ...
      divideEtImpera($P_a, n_a, S_a$)
      *combină* $S_1, ..., S_a$ pentru a obţine $S$

Exemplele tipice de aplicare a acestei metode sunt algoritmii de parcurgere a arborilor binari şi algoritmul de căutare binară.

[1] divide-and-conquer, în engleză

## 12.2 Ordinul de complexitate

Vom presupune că dimensiunea $n_i$ a subproblemei $i$ satisface relaţia $n_i \leq \frac{n}{b}$, unde $b > 1$. Astfel, pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce asigură terminarea subprogramului recursiv.

Presupunem că *divizarea* problemei în subprobleme şi *compunerea* soluţiilor subproblemelor necesită un timp de ordinul $O(n^k)$. Atunci, complexitatea timp $T(n)$ a algoritmului divideEtImpera este dată de relaţia de recurenţă:

$$T(n) = \begin{cases} O(1) & , \text{ dacă } n \leq n_0 \\ a \cdot T(\frac{n}{b}) + O(n^k) & , \text{ dacă } n > n_0 \end{cases} \tag{12.2.1}$$

**Teorema 3** *Dacă $n > n_0$ atunci:*

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \textit{ dacă } a > b^k \\ O(n^k \log_b n) & , \textit{ dacă } a = b^k \\ O(n^k) & , \textit{ dacă } a < b^k \end{cases} \tag{12.2.2}$$

*Demonstraţie:* Putem presupune, fără a restrânge generalitatea, că $n = b^m \cdot n_0$. De asemenea, presupunem că $T(n) = c \cdot n_0^k$ dacă $n \leq n_0$ şi $T(n) = a \cdot T(\frac{n}{b}) + c \cdot n^k$ dacă $n > n_0$. Pentru $n > n_0$ avem:

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + cn^k \\ &= aT\left(b^{m-1}n_0\right) + cn^k \\ &= a\left(aT\left(b^{m-2}n_0\right) + c\left(\frac{n}{b}\right)^k\right) + cn^k \\ &= a^2T\left(b^{m-2}n_0\right) + c\left[a\left(\frac{n}{b}\right)^k + n^k\right] \\ &= ... \\ &= a^mT\left(n_0\right) + c\left[a^{m-1}\left(\frac{n}{b^{m-1}}\right)^k + ... + a\left(\frac{n}{b}\right)^k + n^k\right] \\ &= a^mcn_0^k + c\left[a^{m-1}b^kn_0^k + ... + a\left(b^{m-1}\right)^k n_0^k + (b^m)^k n_0^k\right] \\ &= cn_0^k a^m\left[1 + \frac{b^k}{a} + ... + \left(\frac{b^k}{a}\right)^m\right] \\ &= ca^m\sum_{i=0}^{m}\left(\frac{b^k}{a}\right)^i \end{aligned}$$

unde am notat $cn_0^k$ prin $c$. Distingem cazurile:

1. $a > b^k$. Seria $\sum_{i=0}^{m}\left(\frac{b^k}{a}\right)^i$ este convergentă şi deci şirul sumelor parţiale este convergent. De aici rezultă că $T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$

2. $a = b^k$. Rezultă că $a^m = b^{km} = cn^k$ şi de aici $T(n) = O(n^k m) = O(n^k \log_b n)$.

3. $a < b^k$. Avem $T(n) = O(a^m (\frac{b^k}{a})^m) = O(b^{km}) = O(n^k)$.

## 12.3 Exemple

Dintre problemele clasice care se pot rezolva prin metoda divide et impera menţionăm:

- căutare binară
- sortare rapidă (quickSort)
- problema turnurilor din Hanoi
- sortare prin interclasare
- dreptunghi de arie maximă în placa cu găuri

### 12.3.1 Sortare prin interclasare - MergeSort

Ideea este de a utiliza interclasarea în etapa de asamblare a soluţiilor.

În urma rezolvării recursive a subproblemelor rezultă vectori ordonaţi şi prin interclasarea lor obţinem vectorul iniţial sortat. Vectorii de lungime 1 sunt evident consideraţi sortaţi.

Pasul de divizare se face în timpul $O(1)$. Faza de asamblare se face în timpul $O(m_1 + m_2)$ unde $n_1$ şi $n_2$ sunt lungimile celor doi vectori care se interclasează.

Din Teorema 3 pentru $a = 2, b = 2$ şi $k = 1$ rezultă că ordinul de complexitate al algoritmului MergeSort este $O(n \log_2 n)$ unde $n$ este dimensiunea vectorului iniţial supus sortării.

```
class sort_interclasare
{
  static int x[]={3,5,2,6,4,1,8,2,4,3,5,3};

  public static int[] interclasare(int st,int m,int dr,int a[])
  {
    int i,j,k;
    int b[]=new int[dr-st+1];
    i=st;
    j=m+1;
    k=0;
    while((i<=m)&&(j<=dr))
```

```
    {
      if(a[i]<=a[j]) { b[k]=a[i]; i++;}
         else        { b[k]=a[j]; j++; }
      k++;
    }
    if(i<=m) for(j=i;j<=m; j++) { b[k]=a[j]; k++; }
        else for(i=j;i<=dr;i++) { b[k]=a[i]; k++; }
    k=0;
    for(i=st;i<=dr;i++) { a[i]=b[k];  k++; }
    return a;
  }//interclasare

  public static int[] divide(int st,int dr,int a[])
  {
    int m,aux;
    if((dr-st)<=1)
    {
      if(a[st]>a[dr]) { aux=a[st]; a[st]=a[dr]; a[dr]=aux; }
    }
    else
    {
      m=(st+dr)/2;
      divide(st,m,a);
      divide(m+1,dr,a);
      interclasare(st,m,dr,a);
    }
    return a;
  }

  public static void main(String[] args)
  {
    int i;
    divide(0,x.length-1,x);
    for(i=0;i<x.length;i++) System.out.print(x[i]+" ");
  }//main
}//class
```

### 12.3.2 Placa cu găuri

Se consideră o placă dreptunghiulară în care există $n$ găuri punctiforme de coordonate cunoscute. Să se determine pe această placă un dreptunghi de arie maximă, cu laturile paralele cu laturile plăcii, care să nu conţină găuri în interior ci numai eventual pe laturi.

Vom considera placa într-un sistem cartezian. Considerăm o subproblemă de forma următoare: dreptunghiul determinat de diagonala $(x1, y1)$ (din stânga-jos) şi $(x2, y2)$ din dreapta-sus este analizat pentru a vedea dacă în interior conţine vreo gaură; dacă nu conţine, este o soluţie posibilă (se va alege cea de arie maximă); dacă există o gaură în interiorul său, atunci se consideră patru subprobleme generate de cele 4 dreptunghiuri obţinute prin descompunerea pe orizontală şi pe verticală de cele două drepte paralele cu axele care trec prin punctul care reprezintă gaura.

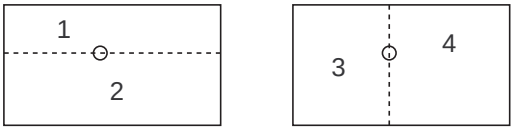



Figura 12.1: Dreptunghi de arie maximă în placa cu găuri

```
import java.io.*;
class drArieMaxima
{
  static int x1,y1,x2,y2,n,x1s,y1s,x2s,y2s,amax;
  static int[] x;
  static int[] y;

  public static void main (String[] args) throws IOException
  {
    int i;
    StreamTokenizer st=new StreamTokenizer(
             new BufferedReader(new FileReader("dreptunghi.in")));
    PrintWriter out=new PrintWriter(
            new BufferedWriter(new FileWriter("dreptunghi.out")));

    st.nextToken(); x1=(int) st.nval;
    st.nextToken(); y1=(int) st.nval;
    st.nextToken(); x2=(int) st.nval;
    st.nextToken(); y2=(int) st.nval;
    st.nextToken(); n=(int) st.nval;

    x=new int [n+1];
    y=new int [n+1];
    for(i=1;i<=n;i++)
    {
      st.nextToken(); x[i]=(int) st.nval;
      st.nextToken(); y[i]=(int) st.nval;
```

```
    }
    dr(x1,y1,x2,y2);
    out.println(amax);
    out.println(x1s+" "+y1s+" "+x2s+" "+y2s);
    out.close();
  }

  static void dr(int x1,int y1,int x2,int y2)
  {
    int i,s=(x2-x1)*(y2-y1);
    if(s<=amax) return;
    boolean gasit=false;
    for(i=1;i<=n;i++)
      if((x1<x[i])&&(x[i]<x2)&&(y1<y[i])&&(y[i]<y2))
      {
        gasit=true;
        break;
      }
    if(gasit1)
    {
      dr(x1,y1,x[i],y2);
      dr(x[i],y1,x2,y2);
      dr(x1,y[i],x2,y2);
      dr(x1,y1,x2,y[i]);
    }
    else
    {
      amax=s;
      x1s=x1;
      y1s=y1;
      x2s=x2;
      y2s=y2;
    }
  }
}
```

# Capitolul 13

# Metoda optimului local - greedy

## 13.1 Metoda greedy

Metoda Greedy are în vedere rezolvarea unor probleme de optim în care optimul global se determină din estimări succesive ale optimului local.

Metoda Greedy se aplică următorului tip de problemă: dintr-o mulţime de elemente $A$, se cere să se determine o submulţime $B$, care îndeplineşte anumite condiţii. De exemplu, alegerea ordinii optime de efectuare a unor lucrări, alegerea traseului optim pentru vizitarea unor obiective turistice, etc. Deoarece este posibil să existe mai multe soluţii se va alege soluţia care maximizează sau minimizează o anumita funcţie obiectiv.

O problemă poate fi rezolvată prin tehnica (metoda) Greedy dacă îndeplineşte proprietatea: dacă $B$ este o soluţie, iar $C$ este inclusă în $B$, atunci şi $C$ este o soluţie.

Pornind de la această condiţie, iniţial se presupune că B este mulţimea vidă şi se adaugă succesiv elemente din $A$ în $B$, ajungând la un optim local. Dar, succesiunea de optimuri locale nu asigură, în general, optimul global. Dacă se demonstrează că succesiunea de optimuri locale conduce la optimul global, atunci metoda Greedy este aplicabilă.

Există următoarele variante ale metodei Greedy:

1. Se pleacă de la soluţia vidă pentru multimea $B$ şi se ia pe rând câte un element din mulţimea $A$. Dacă elementul ales îndeplineşte condiţia de optim local, el este introdus în mulţimea $B$.

2. Se ordonează elementele mulţimii $A$ şi se verifică dacă un element îndeplineşte condiţia de apartenenţă la mulţimea $B$.

## 13.2 Algoritmi greedy

Algoritmii greedy sunt în general simpli şi sunt folosiţi la rezolvarea unor probleme de optimizare. În cele mai multe situaţii de acest fel avem:

- o mulţime de *candidaţi* (lucrări de executat, vârfuri ale grafului, etc.)
- o funcţie care verifică dacă o anumită mulţime de candidaţi constituie o *soluţie posibilă*, nu neapărat *optimă*, a problemei
- o funcţie care verifică dacă o mulţime de candidaţi este *fezabilă*, adică dacă este posibil să completăm această mulţime astfel încat să obţinem o *soluţie posibilă*, nu neapărat *optimă*, a problemei
- o *funcţie de selecţie* care indică la orice moment care este cel mai promiţător dintre candidaţii încă nefolosiţi
- o *funcţie obiectiv* care dă valoarea unei soluţii (timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit, etc) şi pe care urmărim să o optimizăm (minimizăm/maximizăm)

Pentru a rezolva problema de *optimizare*, căutăm o *soluţie posibilă* care să *optimizeze* valoarea *funcţiei obiectiv*.

Un algoritm greedy construieşte soluţia pas cu pas.

Iniţial, mulţimea candidaţilor selectati este vidă.

La fiecare pas, încercăm să adăugăm la această mulţime pe cel mai promiţător candidat, conform *funcţiei de selecţie*. Dacă, după o astfel de adăugare, mulţimea de candidaţi selectaţi nu mai este *fezabilă*, eliminăm ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulţimea de candidaţi selectaţi este *fezabilă*, ultimul candidat adăugat va rămane de acum încolo în ea. De fiecare dată când lărgim mulţimea candidaţilor selectaţi, verificăm dacă această mulţime nu constituie o *soluţie posibilă* a problemei. Dacă algoritmul greedy funcţioneaza corect, prima soluţie găsită va fi totodată o *soluţie optimă* a problemei.

*Soluţia optimă* nu este în mod necesar unică: se poate ca *funcţia obiectiv* să aibă aceeaşi valoare optimă pentru mai multe *soluţii posibile*.

Descrierea formală a unui algoritm greedy general este:

**function** $greedy(C)$ // $C$ este mulţimea candidaţilor
  $S \leftarrow \emptyset$ // $S$ este mulţimea în care construim soluţia
  **while not** $solutie(S)$ **and** $C \neq \emptyset$ **do**
    $x \leftarrow$ un element din $C$ care maximizează/minimizează $select(x)$
    $C \leftarrow C - \{x\}$
    **if** $fezabil(S \cup \{x\})$ **then** $S \leftarrow S \cup \{x\}$
  **if** $solutie(S)$ **then return** $S$
    **else return** "nu există soluţie"

## 13.3 Exemple

Dintre problemele clasice care se pot rezolva prin metoda greedy menţionăm: plata restului cu număr minim de monezi, problema rucsacului, sortare prin selecţie, determinarea celor mai scurte drumuri care pleacă din acelaşi punct (algoritmul lui Dijkstra), determinarea arborelui de cost minim (algoritmii lui Prim şi Kruskal), determinarea mulţimii dominante, problema colorării intervalelor, codificarea Huffman, etc.

### 13.3.1 Plata restului

Un exemplu simplu de algoritm greedy este cel folosit pentru rezolvarea următoarei probleme: trebuie să dăm restul unui client, folosind un număr cât mai mic de monezi. În acest caz, elementele problemei sunt:

- *candidaţii*: mulţimea iniţială de monezi de 1, 5, şi 25 unităţi, pentru care presupunem că avem un număr nelimitat din fiecare tip de monedă
- *o soluţie posibilă*: valoarea totală a unei astfel de mulţimi de monezi selectate trebuie să fie exact valoarea pe care trebuie să o dăm ca rest
- *o mulţime fezabilă*: valoarea totală a monezilor selectate în această mulţime nu este mai mare decât valoarea pe care trebuie să o dăm ca rest
- *funcţia de selecţie*: se alege cea mai mare monedă din mulţimea de candidaţi rămasă
- *funcţia obiectiv*: numărul de monezi folosite în soluţie; se doreşte minimizarea acestui număr

Se poate demonstra că algoritmul greedy va găsi în acest caz mereu soluţia optimă (restul cu un numar minim de monezi).

Pe de altă parte, presupunând că există şi monezi de 12 unităţi sau că unele din tipurile de monezi lipsesc din mulţimea iniţială de candidaţi, se pot găsi destul de uşor contraexemple pentru care algoritmul nu găseşte soluţia optimă, sau nu găseşte nici o soluţie, cu toate că există soluţie.

Evident, soluţia optimă se poate găsi încercând toate combinările posibile de monezi. Acest mod de lucru necesită însa foarte mult timp.

Un algoritm greedy nu duce deci întotdeauna la soluţia optimă, sau la o soluţie. Este doar un principiu general, urmând ca pentru fiecare caz în parte să determinăm dacă obţinem sau nu soluţia optimă.

### 13.3.2 Problema continuă a rucsacului

Se consideră $n$ obiecte. Obiectul $i$ are greutatea $g_i$ şi valoarea $v_i$ $(1 \leq i \leq n)$. O persoană are un rucsac. Fie $G$ greutatea maximă suportată de rucsac. Persoana în cauză doreşte să pună în rucsac obiecte astfel încât valoarea celor din rucsac să fie cât mai mare. Se permite fracţionarea obiectelor (valoarea părţii din obiectul fracţionat fiind direct proporţională cu greutatea ei!).

Notăm cu $x_i \in [0,1]$ partea din obiectul $i$ care a fost pusă în rucsac. Practic, trebuie să maximizăm funcţia

$$f(x) = \sum_{i=1}^{n} x_i c_i.$$

Pentru rezolvare vom folosi metoda greedy. O modalitate de a ajunge la soluţia optimă este de a considera obiectele în ordinea descrescătoare a valorilor utilităţilor lor date de raportul $\frac{v_i}{g_i}$ $(i = 1, ..., n)$ şi de a le încărca întregi în rucsac până când acesta se umple. Din această cauză presupunem în continuare că

$$\frac{v_1}{g_1} \geq \frac{v_2}{g_2} \geq ... \geq \frac{v_n}{g_n}.$$

Vectorul $x = (x_1, x_2, ..., x_n)$ se numeşte *soluţie posibilă* dacă

$$\begin{cases} x_i \in [0,1], \forall i = 1, 2, ..., n \\ \sum_{i=1}^{n} x_i g_i \leq G \end{cases}$$

iar o soluţie posibilă este *soluţie optimă* dacă maximizează funcţia $f$.

Vom nota $G_p$ greutatea permisă de a se încărca în rucsac la un moment dat. Conform strategiei greedy, procedura de rezolvare a problemei este următoarea:

```
procedure rucsac()
    G_p ← G
    for i = 1, n
        if G_p > g_i
            then G_p ← G_p − g_i
            else
                x_i ← G_p / g_i
                for j = i + 1, n
                    x_j ← 0
```

Vectorul $x$ are forma $x = (1, ..., 1, x_i, 0, ..., 0)$ cu $x_i \in (0,1]$ şi $1 \leq i \leq n$.

**Propoziţia 1** *Procedura rucsac() furnizează o soluţie optimă.*

Demonstraţia se găseşte, de exemplu, în [25] la pagina 226.

# Capitolul 14

# Metoda backtracking

Metoda *backtracking* se utilizează pentru determinarea unei submulţimi a unui produs cartezian de forma $S_1 \times S_2 \times ... \times S_n$ (cu mulţimile $S_k$ finite) care are anumite proprietăţi. Fiecare element $(s_1, s_2, ..., s_n)$ al submulţimii produsului cartezian poate fi interpretat ca *soluţie* a unei probleme concrete.

În majoritatea cazurilor nu oricare element al produsului cartezian este soluţie ci numai cele care satisfac anumite *restricţii*. De exemplu, problema determinării tuturor combinărilor de $m$ elemente luate câte $n$ (unde $1 \leq n \leq m$) din mulţimea $\{1, 2, ..., m\}$ poate fi reformulată ca problema determinării submulţimii produsului cartezian $\{1, 2, ..., m\}^n$ definită astfel:

$$\{(s_1, s_2, ..., s_n) \in \{1, 2, ..., m\}^n | s_i \neq s_j, \forall i \neq j, 1 \leq i, j \leq n\}.$$

Metoda se aplică numai atunci când nu există nici o altă cale de rezolvare a problemei propuse, deoarece timpul de execuţie este de ordin exponenţial.

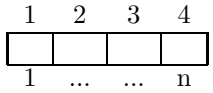
## 14.1 Generarea produsului cartezian

Pentru începători, nu metoda backtracking în sine este dificil de înţeles ci modalitatea de generare a produsului cartezian.

### 14.1.1 Generarea iterativă a produsului cartezian

Presupunem că mulţimile $S_i$ sunt formate din numere întregi consecutive cuprinse între o valoare $min$ şi o valoare $max$. De exemplu, dacă $min = 0, max = 9$ atunci $S_i = \{0, 1, ..., 9\}$. Dorim să generăm produsul cartezian $S_1 \times S_2 \times ... \times S_n$

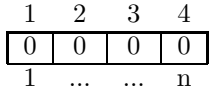
(de exemplu, pentru $n = 4$). Folosim un vector cu $n$ elemente $a = (a_1, a_2, ..., a_n)$ şi vom atribui fiecărei componente $a_i$ valori cuprinse între $min$ şi $max$.



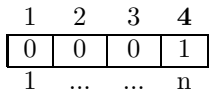

Ne trebuie un *marcaj* pentru a preciza faptul că o anumită componentă este *goală* (nu are plasată în ea o valoare validă). În acest scop folosim variabila `gol` care poate fi iniţializată cu orice valoare întreagă care nu este în intervalul $[min, max]$. Este totuşi utilă iniţializarea `gol=min-1;`.

Cum începem generarea produsului cartezian?

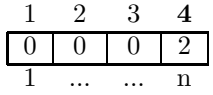
O primă variantă este să atribuim tuturor componentelor $a_i$ valoarea $min$ şi avem astfel un prim element al produsului cartezian, pe care putem să-l afişăm.



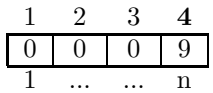

Trebuie să construim următoarele elemente ale produsului cartezian. Este utilă, în acest moment, imaginea kilometrajelor de maşini sau a contoarelor de energie electică, de apă, de gaze, etc. Următoarea configuraţie a acestora este



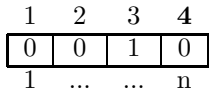

după care urmează




Folosim o variabilă $k$ pentru a urmări poziţia din vector pe care se schimbă valorile. La început $k = n$ şi, pe această poziţie $k$, valorile se schimbă crescând de la $min$ către $max$. Se ajunge astfel la configuraţia ($k = 4 = n$ aici!)



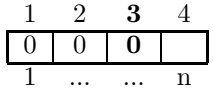

Ce se întâmplă mai departe? Apare configuraţia



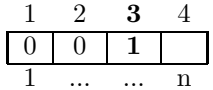

într-un mod ciudat!

Ei bine, se poate spune că aici este cheia metodei backtraching! Dacă reuşim să înţelegem acest pas de trecere de la o configuraţie la alta şi să formalizăm ce am observat, atunci am descoperit singuri această metodă!

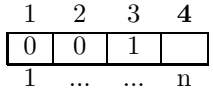
Pe poziţia $k = n$, odată cu apariţia valorii $9 = max$, s-au epuizat toate valorile posibile, aşa că vom considera că poziţia $k$ este goală şi vom trece pe o poziţie mai la stânga, deci $k = k - 1$ (acum $k = 3$).



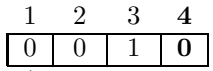
Pe poziţia $k = 3$ se află valoarea 0 care se va schimba cu *următoarea* valoare (dacă există o astfel de valoare $\leq max$), deci în acest caz cu 1.




După plasarea unei valori pe poziţia $k$, se face pasul spre dreapta ($k = k+1$).

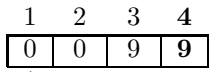



Aici (dacă nu am ieşit în afara vectorului, în urma pasului făcut spre dreapta!), în cazul nostru $k = 4$ şi poziţia este *goală*, se plasează prima valoare validă (deci valoarea $min$).



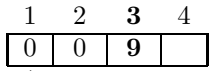

Cum trecerea de la o valoare la alta se face prin creşterea cu o unitate a valorii vechi iar acum facem trecerea `gol` $\rightarrow min$, înţelegem de ce este util să iniţializăm variabila `gol` cu valoarea $min - 1$.

Să considerăm că s-a ajuns la configuraţia



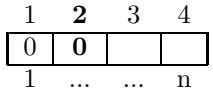

Aici $k = 4 = n$ şi $9 = max$. Pe poziţia $k$ nu se mai poate pune o nouă valoare şi atunci:

- se pune `gol` pe poziţia $k$
- se face pasul spre stânga, deci $k = k - 1$



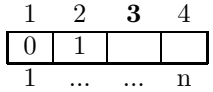

Aici $k = 3$ şi $9 = max$. Pe poziţia $k$ nu se mai poate pune o nouă valoare şi atunci:

- se pune `gol` pe poziţia $k$
- se face pasul spre stânga, deci $k = k - 1$




Aici $k = 2$ şi $0 < max$. Pe poziţia $k$ se poate pune o nouă valoare şi atunci:

- se pune $0 + 1 = 1 =$ *următoarea valoare* pe poziţia $k$
- se face pasul spre dreapta, deci $k = k + 1$




Aici $k = 3$ şi `gol`=-1$< max$. Pe poziţia $k$ se poate pune o nouă valoare:

- se pune `gol`$+1 = -1 + 1 = 0 =$ *următoarea valoare* pe poziţia $k$

• se face pasul spre dreapta, deci $k = k + 1$

0 1 2 3 **4** 5

| 0 | 1 | 0 | |
|---|---|---|---|

0 1 ... ... n n+1

Aici $k = 4$ şi `gol`=-1< $max$. Pe poziţia $k$ se poate pune o nouă valoare:

• se pune `gol`+1= $-1 + 1 = 0$ = *următoarea valoare* pe poziţia $k$

• se face pasul spre dreapta, deci $k = k + 1$

0 1 2 3 4 **5**

| 0 | 1 | 0 | 0 |
|---|---|---|---|

0 1 ... ... n **n+1**

iar aici $\mathbf{k = n + 1}$ şi am ajuns **în afara vectorului!** Nu-i nimic! Se afişează **soluţia** 0100 şi se face pasul la stânga. Aici $k = 4$ şi $0 < max$. Pe poziţia $k$ se poate pune o nouă valoare:

• se pune $0 + 1 = 1$ = *următoarea valoare* pe poziţia $k$

• se face pasul spre dreapta, deci $k = k + 1$

0 1 2 3 4 **5**

| 0 | 1 | 0 | 1 |
|---|---|---|---|

0 1 ... ... n **n+1**

A apărut în mod evident următoarea regulă: ajungând pe poziţia $k \leq n$, încercăm să punem următoarea valoare (`gol` are ca "următoarea valoare" pe $min$) pe această poziţie şi

• dacă se poate: se pune următoarea valoare şi se face pasul spre dreapta;

• dacă nu se poate: se pune `gol`=$min - 1$ şi se face pasul spre stânga.

Presupunem că s-a ajuns la configuraţia

0 1 2 3 4 **5**

| 9 | 9 | 9 | 9 |
|---|---|---|---|

care se afişează ca soluţie. Ajungem la $k = 4$

0 1 ... ... n **n+1**

0 1 2 3 **4** 5

| 9 | 9 | 9 | **9** |
|---|---|---|---|

se goleşte poziţia şi ajungem la $k = 3$

0 1 ... ... n n+1

0 1 2 **3** 4 5

| 9 | 9 | **9** | |
|---|---|---|---|

se goleşte poziţia şi ajungem la $k = 2$

0 1 ... ... n n+1

0 1 **2** 3 4 5

| 9 | **9** | | |
|---|---|---|---|

se goleşte poziţia şi ajungem la $k = 1$

0 1 ... ... n n+1

0 **1** 2 3 4 5

| **9** | | | |
|---|---|---|---|

se goleşte poziţia şi ajungem la $k = 0$

0 1 ... ... n n+1

**0** 1 2 3 4 5

| | | | |
|---|---|---|---|

iar aici $\mathbf{k = 0}$ şi am ajuns **în afara vectorului!**

**0** 1 ... ... n n+1

Nu-i nimic! Aici s-a terminat generarea produsului cartezian!

Putem descrie acum algoritmul pentru generarea produsului cartezian $S^n$, unde $S = \{min, min+1, ..., max\}$:

• structuri de date:

– $a[1..n]$ vectorul soluţie

– $k$ "poziţia" în vectorul soluţie, cu convenţiile $k = 0$ precizează terminarea generării, iar $k = n + 1$ precizează faptul că s-a construit o soluţie care poate fi afişată;

• proces de calcul:

1. se construieşte prima soluţie
2. se plasează poziţia în afara vectorului (în dreapta)
3. cât timp nu s-a terminat generarea
4. dacă este deja construită o soluţie
5. se afişează soluţia şi se face pasul spre stânga
6. altfel, dacă se poate mări valoarea pe poziţia curentă
7. se măreşte cu 1 valoarea şi se face pasul spre dreapta
8. altfel, se goleşte poziţia curentă şi se face pasul spre stânga

3'. revenire la 3.

Implementarea acestui algoritm în Java este următoarea:

```
class ProdCartI1 // 134.000 ms pentru 8 cu 14
{
  static int n=8, min=1, max=14, gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i, k;
    long t1,t2;
    t1=System.currentTimeMillis();
    for(i=1;i<=n;i++) a[i]=min;
    k=n+1;
    while (k>0)
      if (k==n+1) {/* afis(a); */ --k;}
      else { if (a[k]<max) ++a[k++]; else a[k--]=gol; }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}
```

O altă variantă ar putea fi iniţializarea vectorului soluţie cu `gol` şi plasarea pe prima poziţie în vector. Nu mai avem în acest caz o soluţie gata construită dar algoritmul este acelaşi. Implementarea în acest caz este următoarea:

```
class ProdCartI2 // 134.000 ms pentru 8 cu 14
{
  static int n=8, min=1, max=14, gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i, k;
    long t1,t2;
    t1=System.currentTimeMillis();
    for(i=1;i<=n;i++) a[i]=gol;
    k=1;
    while (k>0)
      if (k==n+1) {/* afis(a); */ --k;}
      else { if (a[k]<max) ++a[k++];  else a[k--]=gol; }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}
```

### 14.1.2 Generarea recursivă a produsului cartezian

```
class ProdCartR1 // 101.750 ms pentru 8 cu 14
{
  static int n=8, min=1,max=14;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }
```

```
  static void f(int k)
  {
    int i;
    if (k>n) {/* afis(a); */ return; };
    for(i=min;i<=max;i++) {  a[k]=i;  f(k+1); }
  }

  public static void main (String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}

class ProdCartR2 // 71.300 ms pentru 8 cu 14
{
  static int n=8, min=1,max=14;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  static void f(int k)
  {
    int i;
    for(i=min;i<=max;i++) { a[k]=i; if (k<n) f(k+1); }
  }

  public static void main (String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}
```

```
class ProdCartCar1
{
  static char[] x;
  static int n,m;
  static char[] a={0,'A','X'};

  public static void main(String[] args)
  {
    n=4;
    m=a.length-1;
    x=new char[n+1];
    f(1);
  }

  static void f(int k)
  {
    int i;
    for(i=1;i<=m;i++)
    {
      x[k]=a[i];
      if(k<n) f(k+1);
         else afisv();
    }
  }

  static void afisv()
  {
    int i;
    for(i=1; i<=n; i++)
      System.out.print(x[i]);
    System.out.println();
  }
}//class

class ProdCartCar2
{
  static char[] x;
  static int n;
  static int[] m;
  static char[][] a = { {0},
                        {0,'A','B'},
                        {0,'1','2','3'}
                      };
```

```
  public static void main(String[] args)
  {
    int i;
    n=a.length-1;
    m=new int[n+1];
    for(i=1;i<=n;i++) m[i]=a[i].length-1;
    x=new char[n+1];
    f(1);
  }

  static void f(int k)
  {
    int j;
    for(j=1;j<=m[k];j++)
    {
      x[k]=a[k][j];
      if(k<n) f(k+1); else afisv();
    }
  }

  static void afisv()
  {
    int i;
    for(i=1; i<=n; i++) System.out.print(x[i]);
    System.out.println();
  }
}// class
```

## 14.2 Metoda bactracking

Se foloseşte pentru rezolvarea problemelor care îndeplinesc următoarele condiţii:

1. nu se cunoaşte o altă metodă mai rapidă de rezolvare;
2. soluţia poate fi pusă sub forma unui vector $x = (x_1, x_2, ..., x_n)$ cu $x_i \in A_i$, $i = 1, ..., n$;
3. mulţimile $A_i$ sunt finite.

Tehnica backtracking pleacă de la următoarea premisă:

*dacă la un moment dat nu mai am nici o şansă să ajung la soluţia căutată, renunţ să continui pentru o valoare pentru care ştiu că nu ajung la nici un rezultat.*

Specificul metodei constă în maniera de parcurgere a spaţiului soluţiilor.

• soluţiile sunt construite succesiv, la fiecare etapă fiind completată câte o componentă;

• alegerea unei valori pentru o componentă se face într-o anumită ordine astfel încât să fie asigurată o parcurgere sistematică a spaţiului $A_1 \times A_2 \times ... \times A_n$;

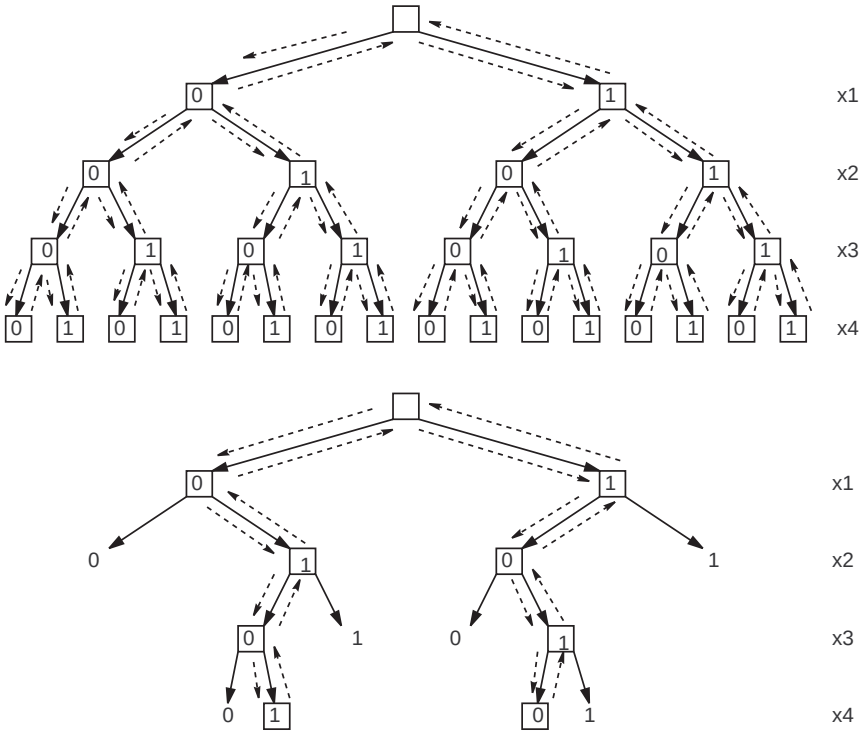
• la completarea componentei $k$ se verifică dacă soluţia parţială $(x_1, x_2, ..., x_k)$ verifică condiţiile induse de restricţiile problemei (acestea sunt numite *condiţii de continuare*);

• dacă au fost încercate toate valorile corespunzătoare componentei $k$ şi încă nu a fost gă sită o soluţie, sau dacše doreşte determinarea unei noi soluţii, se revine la componenta anterioară $(k-1)$ şi se încearcă următoarea valoare corespunză toare acesteia, ş.a.m.d.

• procesul de căutare şi revenire este continuat până când este găsită o soluţie (dacă este suficientă o soluţie) sau până când au foste testate toate configuraţiile posibile (dacă sunt necesare toate soluţiile).

În figură este ilustrat modul de parcurgere a spaţiului soluţiilor în cazul generării produsului cartezian$\{0,1\}^4$.

În cazul în care sunt specificate restricţii (ca de exemplu, să nu fie două componente alăturate egale) anumite ramuri ale structurii arborescente asociate sunt abandonate înainte de a atinge lungimea $n$.



În aplicarea metodei pentru o problemă concretă se parcurg următoarele etape:

• se alege o reprezentare a soluţiei sub forma unui vector cu $n$ componente;

• se identifică mulţimile $A_1, A_2, ..., A_n$ şi se stabileşte o ordine între elemente care să indice modul de parcurgere a fiecărei mulţimi;

• pornind de la restricţiile problemei se stabilesc condiţiile de validitate ale soluţiilor parţiale (condiţiile de continuare).

În aplicaţiile concrete soluţia nu este obţinută numai în cazul în care $k = n$ ci este posibil să fie satisfăcută o condiţie de găsire a soluţiei pentru $k < n$ (pentru anumite probleme nu toate soluţiile conţin acelaşi număr de elemente).

### 14.2.1 Bactracking iterativ

Structura generală a algoritmului este:

| | |
|---|---|
| for(k=1;k<=n;k++) x[k]=gol; | iniţiarizarea vectorului soluţie |
| k=1; | poziţionare pe prima componentă |
| while (k>0) | cât timp există componente de analizat |
| if (k==n+1) | dacă x este soluţie |
| {afis(x); –k;} | atunci: afişare soluţie şi pas stânga |
| else | altfel: |
| { | |
| if(x[k]<max[k]) | dacă există elemente de încercat |
| if(posibil(1+x[k])) | dacă 1+x[k] este validă |
| ++x[k++]; | atunci măresc şi fac pas dreapta |
| else ++x[k]; | altfel măresc şi rămân pe pozitie |
| else x[k–]=gol; | altfel golesc şi fac pas dreapta |
| } | |

Vectorul soluţie $x$ conţine indicii din mulţimile $A_1, A_2, ..., A_n$. Mai precis, $x[k] = i$ înseamnă că pe poziţia $k$ din soluţia $x$ se află $a_i$ din $A_k$.

### 14.2.2 Backtracking recursiv

Varianta recursivă a algoritmului backtracking poate fi realizată după o schemă asemănătoare cu varianta recursivă. Principiul de funcţionare al functiei $f$ (primul apel este $f(1)$) corespunzător unui nivel $k$ este următorul:

– în situaţia în care avem o soluţie, o afişăm şi revenim pe nivelul anterior;

– în caz contrar, se iniţializează nivelul şi se caută un succesor;

– când am găsit un succesor, verificăm dacă este valid. În caz afirmativ, procedura se autoapelează pentru $k + 1$; în caz contrar urmând a se continua căutarea succesorului;

– dacă nu avem succesor, se trece la nivelul inferior $k - 1$ prin ieşirea din procedura recursivă $f$.

## 14.3 Probleme rezolvate

### 14.3.1 Generarea aranjamentelor

*Reprezentarea soluţiilor*: un vector cu $n$ componente.

*Mulţimile* $A_k$: $\{1, 2, ..., m\}$.

*Restricţiile şi condiţiile de continuare*: Dacă $x = (x_1, x_2, ..., x_n)$ este o soluţie ea trebuie să respecte restricţiile: $x_i \neq x_j$ pentru oricare $i \neq j$. Un vector cu $k$ elemente $(x_1, x_2, ..., x_k)$ poate conduce la o soluţie numai dacă satisface condiţiile de continuare $x_i \neq x_j$ pentru orice $i \neq j$, unde $i, j \in \{1, 2, ..., k\}$.

*Condiţia de găsire a unei soluţii*: Orice vector cu $n$ componente care respectă restricţiile este o soluţie. Când $k = n + 1$ a fost găsită o soluţie.

*Prelucrarea soluţiilor*: Fiecare soluţie obţinută este afişată.

```
class GenAranjI1
{
  static int n=2, min=1,max=4, gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++)
      System.out.print(a[i]);
    System.out.println();
  }

  static boolean gasit(int val, int pozmax)
  {
    for(int i=1;i<=pozmax;i++)
      if (a[i]==val) return true;
    return false;
  }

  public static void main (String[] args)
  {
    int i, k;
    for(i=1;i<=n;i++) a[i]=gol;
    k=1;
    while (k>0)
      if (k==n+1) {afis(a); --k;}
      else
```

```
      {
        if(a[k]<max)
          if(!gasit(1+a[k],k-1)) ++a[k++]; // maresc si pas dreapta
            else ++a[k]; // maresc si raman pe pozitie
        else a[k--]=gol;
      }
  }
}

class GenAranjI2
{
  static int n=2, min=1,max=4;
  static int gol=min-1;
  static int[] a=new int[n+1];

  static void afis()
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  static boolean posibil(int k)
  {
    for(int i=1;i<k;i++) if (a[i]==a[k]) return false;
    return true;
  }

  public static void main (String[] args)
  {
    int i, k;
    boolean ok;
    for(i=1;i<=n;i++) a[i]=gol;
    k=1;
    while (k>0)
    {
      ok=false;
      while (a[k] < max) // caut o valoare posibila
      {
        ++a[k];
        ok=posibil(k);
        if(ok) break;
      }
      if(!ok) k--;
      else if (k == n) afis();
```

```
          else a[++k]=0;
    }
  }
}

class GenAranjR1
{
  static int n=2, m=4, nsol=0;
  static int[] a=new int[n+1];

  static void afis()
  {
    System.out.print(++nsol+" : ");
    for(int i=1;i<=n;i++) System.out.print(a[i]+" ");
    System.out.println();
  }

  static void f(int k)
  {
    int i,j;
    boolean gasit;
    for(i=1; i<=m; i++)
    {
      if(k>1) // nu este necesar !
      {
        gasit=false;
        for(j=1;j<=k-1;j++)
          if(i==a[j])
          {
            gasit=true;
            break; // in for j
          }
        if(gasit) continue; // in for i
      }
      a[k]=i;
      if(k<n) f(k+1); else afis();
    }
  }

  public static void main(String[] args)
  {
    f(1);
  }
}// class
```

```
class GenAranjR2
{
  static int[] a;
  static int n,m;

  public static void main(String[] args)
  {
    n=2;
    m=4;
    a=new int[n+1];
    f(1);
  }

  static void f(int k)
  {
    boolean ok;
    int i,j;
    for(i=1;i<=m;i++)
    {
      ok=true; // k=1 ==> nu am in stanga ... for nu se executa !
      for(j=1;j<k;j++)
        if(i==a[j])
        {
          ok=false;
          break;
        }
      if(!ok) continue;
      a[k]=i;
      if(k<n) f(k+1);
        else afisv();
    }
  }

  static void afisv()
  {
    int i;
    for(i=1; i<=n; i++)
       System.out.print(a[i]);
    System.out.println();
  }
}
```

### 14.3.2 Generarea combinărilor

Sunt prezentate mai multe variante (iterative şi recursive) cu scopul de a vedea diferite modalităţi de a câştiga timp în execuţie (mai mult sau mai puţin semnificativ!).

```
class GenCombI1a // 4550 ms cu 12 22 //   40 ms cu 8 14 fara afis
{ // 7640 ms cu 8 14 cu   afis
  static int n=8, min=1,max=14;
  static int gol=min-1,nv=0;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    int i;
    System.out.print(++nv+" : ");
    for(i=1;i<=n;i++)
      System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i, k;
    long t1,t2;
    t1=System.currentTimeMillis();
    for(i=0;i<=n;i++)
      a[i]=gol; // initializat si a[0] care nu se foloseste!!
    k=1;
    while (k>0)
      if (k==n+1) { afis(a);   --k; }
      else
      {
        if(a[k]<max)
          if(1+a[k]>a[k-1])
            ++a[k++];  // maresc si pas dreapta (k>1) ...
          else ++a[k]; // maresc si raman pe pozitie
        else a[k--]=gol;
      }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}// class
```

```
class GenCombI1b // 3825 ms 12 22 sa nu mai merg la n+1 !!!!
{
  static int n=12, min=1,max=22;
  static int gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i;
    long t1,t2;
    t1=System.currentTimeMillis();
    for(i=0;i<=n;i++) a[i]=gol; // initializat si a[0] care nu se foloseste!!
    int k=1;
    while (k>0)
    {
      if(a[k]<max)
        if(1+a[k]>a[k-1])
          if(k<n) ++a[k++]; // maresc si pas dreapta (k>1) ...
          else { ++a[k];  /* afis(a); */}// maresc, afisez  si raman pe pozitie
        else ++a[k];  // maresc si raman pe pozitie
      else a[k--]=gol;
    }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}// class

class GenCombI2a // 1565 ms 12 22
{
  static int n=12, min=1,max=22;
  static int gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }
```

```
  public static void main (String[] args)
  {
    int i, k;
    long t1,t2;
    t1=System.currentTimeMillis();
    for(i=0;i<=n;i++)
      a[i]=gol; // initializat si a[0] care nu se foloseste!!
    k=1;
    while (k>0)
    if (k==n+1) {/* afis(a); */ --k;}
    else
    {
      if(a[k]<max-(n-k)) // optimizat !!!
        if(1+a[k]>a[k-1]) ++a[k++]; // maresc si pas dreapta (k>1) ...
        else ++a[k]; // maresc si raman pe pozitie
      else a[k--]=gol;
    }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}

class GenCombI2b // 1250 ms 12 22
{
  static int n=12, min=1,max=22;
  static int gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++)
      System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i, k; long t1,t2;
    t1=System.currentTimeMillis();
    for(i=0;i<=n;i++)
      a[i]=gol; // initializat si a[0] care nu se foloseste!!
    k=1;
    while (k>0)
```

```
    {
      if(a[k]<max-(n-k)) // optimizat !!!
        if(1+a[k]>a[k-1])
          if(k<n) ++a[k++]; // maresc si pas dreapta (k>1) ...
          else { ++a[k]; /* afis(a); */}// maresc, afisez  si raman pe pozitie
        else ++a[k]; // maresc si raman pe pozitie
      else a[k--]=gol;
    }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}

class GenCombI3a // 835 ms 12 22
{
  static int n=12, min=1, max=22, gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i, k; // si mai optimizat !!!
    long t1,t2;     t1=System.currentTimeMillis();
    for(i=0;i<=n;i++)
      a[i]=i-gol-1; // initializat si a[0] care nu se foloseste!!
    k=1;
    while (k>0)
      if (k==n+1) {/* afis(a); */ --k;}
      else
      {
        if(a[k]<max-(n-k)) // optimizat !!!
          if(1+a[k]>a[k-1]) ++a[k++]; // maresc si pas dreapta (k>1) ...
          else ++a[k]; // maresc si raman pe pozitie
        else a[k--]=k-gol-1; // si mai optimizat !!!
      }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}
```

```
class GenCombI3b // 740 ms 12 22
{
  static int n=12, min=1, max=22, gol=min-1;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  public static void main (String[] args)
  {
    int i, k;
    long t1,t2;
    t1=System.currentTimeMillis();
    for(i=0;i<=n;i++) a[i]=i-gol-1; // si mai optimizat !!!
    k=1;
    while (k>0)
    {
      if(a[k]<max-(n-k)) // optimizat !!!
      {
        ++a[k]; //  maresc
        if(a[k]>a[k-1])
        if(k<n) k++; // pas dreapta
          /* else afis(a); */ // afisez
      }
      else a[k--]=k-gol-1; // si mai optimizat !!!
    }
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}

class GenCombR1a // 2640 ms 12 22
{
  static int[] x;
  static int n,m;

  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    n=12; m=22;
```

```
    x=new int[n+1];
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }

  static void f(int k)
  {
    for(int i=1;i<=m;i++)
    {
      if(k>1) if(i<=x[k-1]) continue;
      x[k]=i;
      if(k<n) f(k+1);/* else afisv(); */
    }
  }

  static void afisv()
  {
    for(int i=1; i<=n; i++) System.out.print(x[i]);
    System.out.println();
  }
}


class GenCombR1b // 2100 ms 12 22
{
  static int n=12,m=22;
  static int[] a=new int[n+1];

  static void afis(int[] a)
  {
    for(int i=1;i<=n;i++) System.out.print(a[i]);
    System.out.println();
  }

  static void f(int k)
  {
    for(int i=1;i<=m;i++ )
    {
      if (i<=a[k-1]) continue; // a[0]=0 deci merge !
      a[k]=i;
      if(k<n) f(k+1); /* else afis(a); */
    }
  }
```

```
  public static void main (String [] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }//main
}//class

class GenCombR2a // 510 ms 12 22
{
  static int n=12, m=22, nsol=0, ni=0; ;
  static int[] x=new int[n+1];

  static void afisx()
  {
    System.out.print(++nsol+" ==> ");
    for(int i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void afis(int k) // pentru urmarirea executiei !
  { // ni = numar incercari !!!
    int i;
    System.out.print(++ni+" : ");
    for(i=1;i<=k;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void f(int k)
  {
    for(int i=k; i<=m-n+k; i++) // imbunatatit !
    {
      if(k>1)
      {
        // afis(k-1);
        if(i<=x[k-1]) continue;
      }
      x[k]=i;
      if(k<n) f(k+1); /* else afisx(); */
    }
  }
```

```
  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}// class

class GenCombR2b // 460 ms 12 22
{
  static int n=12, m=22, nsol=0,ni=0;
  static int[] x=new int[n+1];

  static void afisx()
  {
    System.out.print(++nsol+" ==> ");
    for(int i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void afis(int k) // pentru urmarirea executiei !
  {                       // ni = numar incercari !!!
    System.out.print(++ni+" : ");
    for(int i=1;i<=k;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void f(int k)
  {
    for(int i=k; i<=m-n+k; i++) // imbunatatit !
    {
      if(i<=x[k-1]) continue;
      x[k]=i;
      if(k<n) f(k+1); /* else afisx(); */
    }
  }

  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
```

```
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}// class

class GenCombR3a // 165 ms 12 22
{
  static int n=12;
  static int m=22;
  static int[] x=new int[n+1];
  static int nsol=0,ni=0;

  static void afisx()
  {
    int i;
    System.out.print(++nsol+" ==> ");
    for(i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void afis(int k)
  {
    int i;
    System.out.print(++ni+" : ");
    for(i=1;i<=k;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void f(int k)
  {
    int i;
    for (i=x[k-1]+1; i<=m-n+k; i++) // si mai imbunatatit !!!
    {
      if(k>1)
      {
        // afis(k-1);
        if(i<=x[k-1]) continue;
      }
      x[k]=i;
      if(k<n) f(k+1); /* else afisx(); */
    }
  }
```

```
  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}// class

class GenCombR3b // 140 ms 12 22
{
  static int n=12;
  static int m=22;
  static int[] x=new int[n+1];
  static int nsol=0;

  static void afisx()
  {
    int i;
    System.out.print(++nsol+" : ");
    for(i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void f(int k)
  {
    int i;
    for (i=x[k-1]+1; i<=m-n+k; i++)
    {
      x[k]=i;
      if(k<n) f(k+1); /* else afisx(); */
    }
  }

  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1);
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }
}// class
```

### 14.3.3 Problema reginelor pe tabla de şah

O problemă clasică de generare a configuraţiilor ce respectă anumite restricţii este cea a amplasării damelor pe o tablă de şah astfel încât să nu se atace reciproc.

*Reprezentarea soluţiei*: un vector $x$ unde $x_i$ reprezintă coloana pe care se află regina dacă linia este $i$.

*Restricţii şi condiţii de continuare*: $x_i \neq x_j$ pentru oricare $i \neq j$ (reginele nu se atacă pe coloană) şi $|x_i - x_j| \neq |i - j|$ (reginele nu se atacă pe diagonală).

Sunt prezentate o variantă iterativă şi una recursivă.

```
class RegineI1
{
  static int[] x;
  static int n,nv=0;

  public static void main(String[] args)
  {
    n=5;
    regine(n);
  }

  static void regine(int n)
  {
    int k;
    boolean ok;
    x=new int[n+1];
    for(int i=0;i<=n;i++) x[i]=i;
    k=1;
    x[k]=0;
    while (k>0)
    {
      ok=false;
      while (x[k] <= n-1) // caut o valoare posibila
      {
        ++x[k];
        ok=posibil(k);
        if(ok) break;
      }
      if(!ok) k--;
      else if (k == n) afis();
           else x[++k]=0;
    }
  }//regine()
```

```
  static boolean posibil(int k)
  {
    for(int i=1;i<=k-1;i++)
      if ((x[k]==x[i])||((k-i)==Math.abs(x[k]-x[i]))) return false;
    return true;
  }

  static void afis()
  {
    int i;
    System.out.print(++nv+" : ");
    for(i=1; i<=n; i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}// class


class RegineR1
{
  static int[] x;
  static int n,nv=0;

  public static void main(String[] args)
  {
    n=5;
    x=new int[n+1];
    f(1);
  }

  static void f(int k)
  {
    boolean ok;
    int i,j;
    for(i=1;i<=n;i++)
    {
      ok=true;
      for(j=1;j<k;j++)
      if((i==x[j])||((k-j)==Math.abs(i-x[j]))) {ok=false; break;}
      if(!ok) continue;
      x[k]=i;
      if(k<n) f(k+1); else afisv();
    }
  }
```

```
  static void afisv()
  {
    int i;
    System.out.print(++nv+" : ");
    for(i=1; i<=n; i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}
```

### 14.3.4 Turneul calului pe tabla de şah

```
import java.io.*;
class Calut
{
  static final int  LIBER=0,SUCCES=1,ESEC=0,NMAX=8;
  static final int[]  a={0,2,1,-1,-2,-2,-1,1,2}; // miscari posibile pe Ox
  static final int[] b={0,1,2,2,1,-1,-2,-2,-1}; // miscari posibile pe Oy
  static int[][] tabla=new int[NMAX+1][NMAX+1]; // tabla de sah
  static int n,np,ni=0; // np=n*n

  public static void main(String[] args) throws IOException
  {
    BufferedReader br=new BufferedReader(
                      new InputStreamReader(System.in));
    while ((n<3)||(n>NMAX))
    {
      System.out.print("Dimensiunea tablei de sah [3.."+NMAX+"] : ");
      n=Integer.parseInt(br.readLine());
    }
    np=n*n;
    for(int i=0;i<=n;i++)
      for(int j=0;j<=n;j++) tabla[i][j]=LIBER;
    tabla[1][1]=1;
    if(incerc(2,1,1)==SUCCES) afisare();
      else System.out.println("\nNu exista solutii !!!");
    System.out.println("\n\nNumar de incercari = "+ni);
  } // main

  static void afisare()
  {
    System.out.println("\r---------------------------------");
```

```
    for(int i=1;i<=n;i++)
    {
      System.out.println();
      for(int j=1;j<=n;j++) System.out.print("\t"+tabla[i][j]);
    }
  }// afisare()

  static int incerc(int i, int x, int y)
  {
    int xu,yu,k,rezultatIncercare; ni++;
    k=1;
    rezultatIncercare=ESEC;
    while ((rezultatIncercare==ESEC)&&(k<=8))// miscari posibile cal
    {
      xu=x+a[k]; yu=y+b[k];
      if((xu>=1)&&(xu<=n)&&(yu>=1)&&(yu<=n))
        if(tabla[xu][yu]==LIBER)
        {
          tabla[xu][yu]=i;
          // afisare();
          if (i<np)
          {
            rezultatIncercare=incerc(i+1,xu,yu);
            if(rezultatIncercare==ESEC) tabla[xu][yu]=LIBER;
          }
          else rezultatIncercare=SUCCES;
        }
        k++;
    }// while
    return rezultatIncercare;
  }// incerc()
}// class
```

Pe ecran apar următoarele rezultate:

```
Dimensiunea tablei de sah [3..8] : 5
---------------------------------

        1       6       15      10      21
        14      9       20      5       16
        19      2       7       22      11
        8       13      24      17      4
        25      18      3       12      23

Numar de incercari = 8839
```

### 14.3.5 Problema colorării hărţilor

Se consideră o hartă cu $n$ ţări care trebuie colorată folosind $m < n$ culori, astfel încât oricare două ţări vecine să fie colorate diferit. Relaţia de vecinătate dintre ţări este reţinută într-o matrice $n \times n$ ale cărei elemente sunt:

$$v_{i,j} = \begin{cases} 1 & \text{dacă i este vecină cu j} \\ 0 & \text{dacă i nu este vecină cu j} \end{cases}$$

*Reprezentarea soluţiilor*: O soluţie a problemei este o modalitate de colorare a ţă rilor şi poate fi reprezentată printru-un vector $x = (x_1, x_2, ..., x_n)$ cu $x_i \in \{1, 2, ..., m\}$ reprezentând culoarea asociată ţării $i$. Mulţimile de valor ale elementelor sint $A_1 = A_2 = ... = A_n = \{1, 2, ..., m\}$.

*Restricţii şi condiţii de continuare*: Restricţia ca două ţări vecine să fie colorate diferit se specifică prin: $x_i \neq x_j$ pentru orice $i$ şi $j$ având proprietatea $v_{i,j} = 1$. Condiţia de continuare pe care trebuie să o satisfacă soluţia parţială $(x_1, x_2, ..., x_k)$ este: $x_k \neq x_i$ pentru orice $i < k$ cu proprietatea că $v_{i,k} = 1$.

```
class ColorareHartiI1
{
  static int nrCulori=3;// culorile sunt 1,2,3
  static int[][] harta=
                       {//  CoCaIaBrTu
                        {2,1,1,1,1},// Constanta  (0)
                        {1,2,1,0,0},// Calarasi  (1)
                        {1,1,2,1,0},// Ialomita  (2)
                        {1,0,1,2,1},// Braila  (3)
                        {1,0,0,1,2} // Tulcea  (4)
                       };
  static int n=harta.length;
  static int[] culoare=new int[n];
  static int nrVar=0;

  public static void main(String[] args)
  {
    harta();
    if(nrVar==0)
      System.out.println("Nu se poate colora !");
  } // main

  static void harta()
  {
    int k; // indicele pentru tara
    boolean okk;
```

```
    k=0; // prima pozitie
    culoare[k]=0; // tara k nu este colorata (inca)
    while (k>-1) // -1 = iesit in stanga !!!
    {
      okk=false;
      while(culoare[k] < nrCulori)// selectez o culoare
      {
        ++culoare[k];
        okk=posibil(k);
        if (okk) break;
      }
      if  (!okk)
        k--;
        else if (k == (n-1))
               afis();
               else culoare[++k]=0;
    }
  } // harta

  static boolean posibil(int k)
  {
    for(int i=0;i<=k-1;i++)
    if((culoare[k]==culoare[i])&&(harta[i][k]==1))
      return false;
    return true;
  } // posibil

  static void afis()
  {
    System.out.print(++nrVar+" : ");
    for(int i=0;i<n;i++)
      System.out.print(culoare[i]+" ");
    System.out.println();
  } // scrieRez
} // class
```

Pentru 3 culori rezultatele care apar pe ecran sunt:

```
1       1 2 3 2 3
2       1 3 2 3 2
3       2 1 3 1 3
4       2 3 1 3 1
5       3 1 2 1 2
6       3 2 1 2 1
```

```
class ColorareHartiR1
{
  static int[] x;
  static int[][] a=
                  {
                   {0,0,0,0,0,0},
                   {0,2,1,1,1,1},// Constanta  (1)
                   {0,1,2,1,0,0},// Calarasi  (2)
                   {0,1,1,2,1,0},// Ialomita  (3)
                   {0,1,0,1,2,1},// Braila  (4)
                   {0,1,0,0,1,2} // Tulcea  (5)
                  };
  static int n,m,nv=0;

  public static void main(String[] args)
  {
    n=a.length-1;
    m=3; // nr culori
    x=new int[n+1];
    f(1);
  }

  static void f(int k)
  {
    boolean ok;
    int i,j;
    for(i=1;i<=m;i++)
    {
      ok=true;
      for(j=1;j<k;j++)
        if((i==x[j])&&(a[j][k]==1)) {ok=false; break;}
      if(!ok) continue;
      x[k]=i;
      if(k<n) f(k+1); else afisv();
    }
  }

  static void afisv()
  {
    System.out.print(++nv+" : ");
    for(int i=1; i<=n; i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}
```

### 14.3.6 Problema vecinilor

Un grup de $n$ persoane sunt aşezate pe un rând de scaune. Între oricare doi vecini izbucnesc conflicte. Rearanjaţi persoanele pe scaune astfel încât între oricare doi vecini "certaţi" să existe una sau cel mult două persoane cu care nu au apucat să se certe! Afişaţi toate variantele de reaşezare posibile.

Vom rezolva problema prin metada backtracking. Presupunem că persoanele sunt numerotate la început, de la stanga la dreapta cu $1, 2, ..., n$. Considerăm ca soluţie un vector $x$ cu $n$ componente pentru care $x_i$ reprezintă "poziţia pe care se va afla persoana $i$ după reaşezare".

Pentru $k > 1$ dat, condiţiile de continuare sunt:

a) $x_j \neq x_k$, pentru $j = 1, ..., k-1$ ($x$ trebuie să fie permutare)

b) $|x_k - x_{k-1}| = 2$ sau 3 (între persoana $k$ şi vecinul său anterior trebuie să se afle una sau două persoane)

În locul solutiei $x$ vom lista permutarea $y = x^{-1}$ unde $y_i$ reprezintă persoana care se asează pe locul $i$.

```
class VeciniR1
{
  static int[] x;
  static int n,m,nsol=0;

  public static void main(String[] args)
  {
    n=7, m=7;
    x=new int[n+1];
    f(1);
  }

  static void f(int k)
  {
    boolean ok;
    int i,j;
    for(i=1;i<=m;i++)
    {
      ok=true;
      for(j=1;j<k;j++) if(i==x[j]) {ok=false; break;}
      if(!ok) continue;
      if((Math.abs(i-x[k-1])!=2)&&(Math.abs(i-x[k-1])!=3)) continue;
      x[k]=i;
      if(k<n) f(k+1); else afis();
    }
```

```
  }

  static void afis()
  {
    int i;
    System.out.print(++nsol+" : ");
    for(i=1; i<=n; i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}


import java.io.*;
class Vecini
{
  static int nrVar=0,n;
  static int[] x,y;

  public static void main(String[] args) throws IOException
  {
    BufferedReader br=new BufferedReader(
                      new InputStreamReader(System.in));
    while ((n<3)||(n>50))
    {
      System.out.print("numar persoane [3..50] : ");
      n=Integer.parseInt(br.readLine());
    }
    x=new int[n];
    y=new int[n];
    reasez(0);
    if(nrVar==0) System.out.println("Nu se poate !!!");
  } // main

  static void reasez(int k)
  {
    int i,j;
    boolean ok;
    if (k==n) scrieSolutia();// n=in afara vectorului !!!
    else for(i=0;i<n;i++)
    {
      ok=true;
      j=0;
      while ((j<k-1) && ok) ok=(i != x[j++]);
      if (k>0)
```

```
        ok=(ok&&((Math.abs(i-x[k-1])==2)||(Math.abs(i-x[k-1])==3)));
      if (ok) { x[k]=i; reasez(k+1);}
    }
  } // reasez

  static void scrieSolutia()
  {
    int i;
    for(i=0;i<n;i++) y[x[i]]=i;// inversarea permutarii !!!
    System.out.print(++nrVar+" : ");
    for(i=0;i<n;i++)
      System.out.print(++y[i]+" "); // ca sa nu fie 0,1,...,n-1
    System.out.println();           // ci 1,2,...,n (la afisare)
  } // scrieRez
} // class
```

### 14.3.7 Problema labirintului

Se dă un labirint sub formă de matrice cu $m$ linii şi $n$ coloane. Fiecare element al matricei reprezintă o cameră a labirintului. Într-una din camere, de coordonate $x_0$ şi $y_0$ se gă seşte un om. Se cere să se găească toate ieşirile din labirint.

O primă problemă care se pune este precizarea modului de codificare a ieşirilor din fiecare cameră a labirintului.

Dintr-o poziţie oarecare $(i, j)$ din labirint, deplasarea se poate face în patru direcţii: Nord, Est, Sud şi Vest considerate în această ordine (putem alege oricare altă ordine a celor patru direcţii, dar odată aleasă aceasta se păstrează până la sfârşitul problemei).

Fiecare cameră din labirint poate fi caracterizată printr-un şir de patru cifre binare asociate celor patru direcţii, având semnificaţie faptul că acea cameră are sau nu ieşiri pe direcţiile considerate.

De exemplu, dacă pentru camera cu poziţia $(2, 4)$ există ieşiri la N şi S, ei îi va corespunde şirul 1010 care reprezintă numărul 10 în baza zece.

Prin urmare, codificăm labirintul printr-o matrice $a[i][j]$ cu elemente între 1 şai 15. Pentru a testa uşor ieşirea din labirint, matricea se bordează cu două linii şi două coloane de valoare egală cu 16.

Ne punem problema determinării ieşirilor pe care le are o cameră.

O cameră are ieşirea numai spre N dacă şi numai dacă $a[i][j]\&\&8 \neq 0$.

Drumul parcurs la un moment dat se reţine într-o matrice cu două linii, $d$, în care:

– $d[1][k]$ reprezintă linia camerei la care s-a ajuns la pasul $k$;

– $d[2][k]$ reprezintă coloana camerei respective.

La găsirea unei ieşiri din labirint, drumul este afişat.

Principiul algoritmului este următorul:

– se testează dacă s-a ieşit din labiritn (adică $a[i][j] = 16$);
– în caz afirmativ se afişează drumul găsit;
– în caz contrar se procedează astfel:
• se reţin în matricea $d$ coordonatele camerei vizitate;
• se verifică dacă drumul arcurs a mai trecut prin această cameră, caz în care se iese din procedură;
• se testează pe rând ieşirile spre N, E, S, V şi acolo unde este găsită o astfel de ieşire se reapelează procedura cu noile coordonate;
• înaintea ieşirii din procedură se decrementează valoarea lui $k$.

```
import java.io.*;
class Labirint
{
  static final char coridor='.', start='x',
                    gard='H',    pas='*',  iesire='E';
  static char[][] l;
  static int m,n,x0,y0;
  static boolean ok;

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    StreamTokenizer st = new StreamTokenizer(
                         new BufferedReader(new FileReader("labirint.in")));
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;
    l=new char[m][n];
    for(i=0;i<m;i++)
    {
      st.nextToken();
      for(j=0;j<n;j++) l[i][j]=st.sval.charAt(j);
    }

    st.nextToken(); x0=(int)st.nval;
    st.nextToken(); y0=(int)st.nval;

    ok=false;
    gi(x0,y0);
    l[x0][y0]=start;
    PrintWriter out = new PrintWriter(
                      new BufferedWriter(new FileWriter("labirint.out")));
    for(i=0;i<m;i++)
    {
      if (i>0) out.println();
```

```
      for(j=0;j<n;j++) out.print(l[i][j]);
    }
    if (!ok) out.println("NU exista iesire !");
    out.close();
  }// main()

  static void gi(int x,int y)
  {
    if ((x==0)||(x==n-1)||(y==0)||(y==m-1)) ok=true;
    else
    {
      l[x][y]=pas;
      if(l[x][y+1]==coridor||l[x][y+1]==iesire) gi(x,y+1);
      if(!ok&&(l[x+1][y]==coridor||l[x+1][y]==iesire)) gi(x+1,y);
      if(!ok&&(l[x][y-1]==coridor||l[x][y-1]==iesire)) gi(x,y-1);
      if(!ok&&(l[x-1][y]==coridor||l[x-1][y]==iesire)) gi(x-1,y);
      l[x][y]=coridor;
    }
    if (ok) l[x][y]=pas;
  }
}// class
```

De exemplu, pentru fisierul de intrare: labirint.in

```
8 8
HHHHHHEH
H....H.H
H.HHHH.H
H.HHHH.H
H....H.H
H.HHHH.H
H......H
HHHHHHEH
2 2
```

se obţine fişierul de ieşire: labirint.out

```
HHHHHHEH
H....H.H
H*xHHH.H
H*HHHH.H
H*...H.H
H*HHHH.H
H******H
HHHHHH*H
```

### 14.3.8 Generarea partiţiilor unui număr natural

Să se afişeze toate modurile de descompunere a unui număr natural $n$ ca sumă de numere naturale.

Vom folosi o procedură $f$ care are doi parametri: componenta la care s-a ajuns ($k$) şi o valoare $v$ (care conţine diferenţa care a mai rămas până la $n$).

Iniţial, procedura este apelată pentru nivelul 1 şi valoarea $n$. Imediat ce este apelată, procedura va apela o alta pentru afişarea vectorului (iniţial afişează $n$).

Din valoarea care se găseşte pe un nivel, $S[k]$, se scad pe rând valorile $1, 2, ..., S[k] - 1$, valori cu care se apelează procedura pentru nivelul următor.

La revenire se reface valoarea existentă.

```
class PartitieNrGenerare // n = suma de numere
{
  static int dim=0, nsol=0, n=6;
  static int[] x=new int[n+1];

  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(n,n,1);
    t2=System.currentTimeMillis();
    System.out.println("nsol = "+nsol+" timp = "+(t2-t1)+"\n");
  }

  static void f(int val, int maxp, int poz)
  {
    if(maxp==1) { nsol++; dim=poz-1; afis2(val,maxp); return; }
    if(val==0)  { nsol++; dim=poz-1; afis1();         return; }
    int maxok=min(maxp,val);
    for(int i=maxok;i>=1;i--)
    {
      x[poz]=i;
      f(val-i,min(val-i,i),poz+1);
    }
  }

  static void afis1()
  {
    System.out.print("\n"+nsol+" : ");
    for(int i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }
```

```
  static void afis2(int val,int maxip)
  {
    int i;
    System.out.print("\n"+nsol+" : ");
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
    for(i=1;i<=val;i++) System.out.print("1 ");
  }

  static int min(int a,int b) { return (a<b)?a:b; }
}
```

Pentru descompunerea ca sumă de numere impare, programul este:

```
class PartitieNrImpare1 // n = suma de numere impare
{ // nsol = 38328320 1Gata!!!  timp = 8482
  static int dim=0,nsol=0;
  static int[] x=new int[161];

  public static void main(String[] args)
  {
    long t1,t2;
    int n=160, maxi=((n-1)/2)*2+1;
    t1=System.currentTimeMillis();
    f(n,maxi,1);
    t2=System.currentTimeMillis();
    System.out.println("nsol = "+nsol+" timp = "+(t2-t1)+"\n");
  }

  static void f(int val, int maxip, int poz)
  {
    if(maxip==1)
    {
      nsol++;
      // dim=poz-1;
      // afis2(val,maxip);
     return;
    }
    if(val==0)
    {
      nsol++;
      // dim=poz-1;  afis1();
      return;
    }
```

```
    int maxi=((val-1)/2)*2+1;
    int maxiok=min(maxip,maxi);
    for(int i=maxiok;i>=1;i=i-2)
    {
      x[poz]=i;
      f(val-i,min(maxiok,i),poz+1);
    }
  }

  static void afis1()
  {
    System.out.print("\n"+nsol+" : ");
    for(int i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }

  static void afis2(int val,int maxip)
  {
    int i;
    System.out.print("\n"+nsol+" : ");
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
    for(i=1;i<=val;i++) System.out.print("1 ");
  }

  static int max(int a,int b) { return (a>b)?a:b; }
  static int min(int a,int b) { return (a<b)?a:b; }
}// class
```

O versiune optimizată este:

```
class PartitieNrImpare2 // n = suma de numere impare ;
{   // optimizat timp la jumatate !!!
  static int dim=0,nsol=0; // nsol = 38328320 2Gata!!!  timp = 4787
  static int[] x=new int[161];

  public static void main(String[] args)
  {
    long t1,t2;
    int n=160, maxi=((n-1)/2)*2+1;
    t1=System.currentTimeMillis();
    f(n,maxi,1);
    t2=System.currentTimeMillis();
    System.out.println("nsol = "+nsol+" timp= "+(t2-t1));
  }
```

```
  static void f(int val, int maxip, int poz)
  {
    if(maxip==1)
    {
      nsol++;
      // dim=poz-1;  afis2(val);
      return;
    }
    if(val==0)
    {
      nsol++;
      // dim=poz-1;  afis1();
      return;
    }

    int maxi=((val-1)/2)*2+1;
    int maxiok=min(maxip,maxi);
    for(int i=maxiok;i>=3;i=i-2)
    {
      x[poz]=i;
      f(val-i,i,poz+1);
    }
    nsol++;
    // dim=poz-1;
    // afis2(val);
  }

  static void afis1()
  {
    System.out.print("\n"+nsol+" : ");
    for(int i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }

  static void afis2(int val)
  {
    System.out.print("\n"+nsol+" : ");
    for(int i=1;i<=dim;i++) System.out.print(x[i]+" ");
    for(int i=1;i<=val;i++) System.out.print("1 ");
  }

  static int max(int a,int b) { return (a>b)?a:b; }
  static int min(int a,int b) { return (a<b)?a:b; }
}// class
```

### 14.3.9 Problema parantezelor

Problema cere generarea tuturor combinaţiilor de $2n$ paranteze ($n$ paranteze de deschidere şi $n$ paranteze de închidere) care se închid corect.

```
class ParantezeGenerare // 2*n paranteze
{
  static int nsol=0;
  static int n=4;
  static int n2=2*n;
  static int[] x=new int[n2+1];

  public static void main(String[] args)
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    f(1,0,0);
    t2=System.currentTimeMillis();
    System.out.println("nsol = "+nsol+" timp = "+(t2-t1)+"\n");
  }

  static void f(int k, int npd, int npi)
  {
    if(k>n2) afis();
    else
    {
      if(npd<n)   { x[k]=1; f(k+1,npd+1,npi); }
      if(npi<npd) { x[k]=2; f(k+1,npd,npi+1); }
    }
  }

  static void afis()
  {
    int k;
    System.out.print(++nsol+" : ");
    for(k=1;k<=n2;k++)
      if(x[k]==1) System.out.print("( ");
      else System.out.print(") ");
    System.out.println();
  }
}// class
```

# Capitolul 15

# Programare dinamică

## 15.1 Prezentare generală

Folosirea tehnicii programării dinamice solicită experienţă, intuiţie şi abilităţi matematice. De foarte multe ori rezolvările date prin această tehnică au ordin de complexitate polinomial.

În literatura de specialitate există două variante de prezentare a acestei tehnici. Prima dintre ele este mai degrabă de natură deductivă pe când a doua foloseşte gândirea inductivă.

Prima variantă se bazează pe conceptul de *subproblemă*. Sunt considerate următoarele aspecte care caracterizează o rezolvare prin programare dinamică:

• Problema se poate descompune recursiv în mai multe *subprobleme* care sunt caracterizate de *optime parţiale*, iar *optimul global* se obţine prin *combinarea* acestor *optime parţiale*.

• *Subproblemele* respective *se suprapun*. La un anumit nivel, două sau mai multe subprobleme necesită rezolvarea unei aceeaşi subprobleme. Pentru a evita risipa de timp rezultată în urma unei implementări recursive, optimele parţiale se vor reţine treptat, în maniera bottom-up, în anumite structuri de date (tabele).

A doua variantă de prezentare face apel la conceptele intuitive de sistem, stare şi decizie. O problemă este abordabilă folosind tehnica programării dinamice dacă satisface *principiul de optimalitate* sub una din formele prezentate în continuare.

Fie secvenţa de stări $S_0, S_1, ..., S_n$ ale sistemului.

• Dacă $d_1, d_2, ..., d_n$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_0$ în starea $S_n$, atunci pentru orice $i$ $(1 \leq i \leq n)$ $d_{i+1}, d_{i+2}, ..., dn$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_i$ în starea $S_n$.

Astfel, decizia $d_i$ depinde de deciziile anterioare $d_{i+1}, ..., d_n$. Spunem că se aplică *metoda înainte*.

• Dacă $d_1, d_2, ..., d_n$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_0$ în starea $S_n$, atunci pentru orice $i$ $(1 \leq i \leq n)$ $d_1, d_2, ..., d_i$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_0$ în starea $S_i$. Astfel, decizia $d_{i+1}$ depinde de deciziile anterioare $d_1, ..., d_i$. Spunem că se aplică *metoda înapoi*.

• Dacă $d_1, d_2, ..., d_n$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_0$ în starea $S_n$, atunci pentru orice $i$ $(1 \leq i \leq n)$ $d_{i+1}, d_{i+2}, ..., d_n$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_i$ în starea $S_n$ şi $d_1, d_2, ..., d_i$ este un *şir optim de decizii* care duc la trecerea sistemului din starea $S_0$ în starea $S_i$. Spunem că se aplică *metoda mixtă*.

Indiferent de varianta de prezentare, rezolvarea prin programare dinamică presupune găsirea şi rezolvarea unui sistem de recurenţe. În prima variantă avem recurenţe între subprobleme, în a doua variantă avem recurenţe în şirul de decizii. În afara cazurilor în care recurenţele sunt evidente, este necesară şi o justificare sau o demonstraţie a faptului că aceste recurenţe sunt cele corecte.

Deoarece rezolvarea prin recursivitate duce de cele mai multe ori la ordin de complexitate exponenţial, se folosesc tabele auxiliare pentru reţinerea optimelor parţiale iar spaţiul de soluţii se parcurge în ordinea crescătoare a dimensiunilor subproblemelor. Folosirea acestor tabele dă şi numele tehnicii respective.

Asemănător cu metoda "divide et impera", programarea dinamică rezolvă problemele combinând soluţiile unor subprobleme. Deosebirea constă în faptul că "divide et impera" partiţionează problema în subprobleme independente, le rezolvă (de obicei recursiv) şi apoi combină soluţiile lor pentru a rezolva problema initială, în timp ce programarea dinamică se aplică problemelor ale căror subprobleme nu sunt independente, ele având "sub-subprobleme" comune. În această situaţie, se rezolvă fiecare "sub-subproblemă" o singură dată şi se foloseşte un tablou pentru a memora soluţia ei, evitându-se recalcularea ei de câte ori subproblema reapare.

Programarea dinamică se aplică problemelor care au mai multe soluţii, fiecare din ele cu câte o valoare, şi la care se doreşte obţinerea unei soluţii optime (minime sau maxime).

Algoritmul pentru rezolvarea unei probleme folosind programarea dinamică se dezvolta în 4 etape:

1. caracterizarea unei soluţii optime (identificarea unei modalităti optime de rezolvare, care satisface una dintre formele principiului optimalităţii)

2. definirea recursivă a valorii unei soluţii optime (dar nu se implementează recursiv)

3. calculul efectiv al valorii unei soluţii optime folosind o structură de date (de obicei tablou)

4. reconstituirea unei soluţii pe baza informaţiei calculate.

## 15.2 Probleme rezolvate

### 15.2.1 Inmulţirea optimală a matricelor

Considerăm $n$ matrice $A_1, A_2, ..., A_n$, de dimensiuni $d_0 \times d_1$, $d_1 \times d_2$, ..., $d_{n-1} \times d_n$. Produsul $A_1 \times A_2 \times ... \times A_n$ se poate calcula în diverse moduri, aplicând asociativitatea operaţiei de înmultire a matricelor.

Numim *înmulţire elementară* înmulţirea a două elemente.

În funcţie de modul de parantezare diferă numărul de înmulţiri elementare necesare pentru calculul produsului $A_1 \times A_2 \times ... \times A_n$.

Se cere parantezare optimală a produsului $A_1 \times A_2 \times ... \times A_n$ (pentru care *costul*, adică numărul total de înmulţiri elementare, să fie minim).

*Exemplu*:

Pentru 3 matrice de dimensiuni $(10, 1000)$, $(1000, 10)$ şi $(10, 100)$, produsul $A_1 \times A_2 \times A_3$ se poate calcula în două moduri:

1. $(A_1 \times A_2) \times A_3$ necesitând 1000000+10000=1010000 înmulţiri elementare
2. $A_1 \times (A_2 \times A_3)$, necesitând 1000000+1000000=2000000 înmulţiri.

Reamintim că numărul de înmulţiri elementare necesare pentru a înmulţi o matrice $A$, având $n$ linii şi $m$ coloane, cu o matrice $B$, având $m$ linii şi $p$ coloane, este $n * m * p$.

*Solutie*

1. Pentru a calcula $A_1 \times A_2 \times ... \times A_n$, în final trebuie să înmulţim două matrice, deci vom paranteza produsul astfel: $(A_1 \times A_2 \times ... \times A_k) \times (A_{k+1} \times ... \times A_n)$. Această observaţie se aplică şi produselor dintre paranteze. Prin urmare, subproblemele problemei iniţiale constau în determinarea parantezării optimale a produselor de matrice de forma $A_i \times A_{i+1} \times ... \times A_j$, $1 \leq i \leq j \leq n$. Observăm că subproblemele nu sunt independente. De exemplu, calcularea produsului $A_i \times A_{i+1} \times ... \times A_j$ şi calcularea produsului $A_{i+1} \times A_{i+2} \times ... \times A_{j+1}$, au ca subproblemă comună calcularea produsului $A_{i+1} \times ... \times A_j$.

2. Pentru a reţine soluţiile subproblemelor, vom utiliza o matrice $M$, cu $n$ linii şi $n$ coloane, cu semnificatia:

   $M[i][j]$ = numărul minim de înmulţiri elementare necesare pentru a calcula produsul $A_i \times A_{i+1} \times ... \times A_j$, $1 \leq i \leq j \leq n$.

   Evident, numărul minim de înmulţiri necesare pentru a calcula $A_1 \times A_2 \times ... \times A_n$ este $M[1][n]$.

3. Pentru ca parantezarea să fie optimală, parantezarea produselor $A_1 \times A_2 \times ... \times A_k$ şi $A_{k+1} \times ... \times A_n$ trebuie să fie de asemenea optimală. Prin urmare elementele matricei $M$ trebuie să satisfacă urmatoarea relaţie de recurenţă:

$$M[i][i] = 0, i = 1, 2, ..., n.$$

$$M[i][j] = \min_{i \leq k < j} \{M[i][k] + M[k+1][j] + d[i-1] \times d[k] \times d[j]\}$$

Cum interpretăm această relaţie de recurenţă? Pentru a determina numărul minim de înmulţiri elementare pentru calculul produsului $A_i \times A_{i+1} \times ... \times A_j$, fixăm poziţia de parantezare $k$ în toate modurile posibile (între $i$ şi $j-1$), şi alegem varianta care ne conduce la minim. Pentru o poziţie $k$ fixata, costul parantezării este egal cu numarul de înmulţiri elementare necesare pentru calculul produsului $A_i \times A_{i+1} \times ... \times A_k$, la care se adaugă numărul de înmulţiri elementare necesare pentru calculul produsului $A_{k+1} \times ... \times A_j$ şi costul înmulţirii celor două matrice rezultate ($d_{i-1} \times d_k \times d_j$).

Observăm că numai jumătatea de deasupra diagonalei principale din $M$ este utilizată. Pentru a construi soluţia optimă este utilă şi reţinerea indicelui $k$, pentru care se obtine minimul. Nu vom considera un alt tablou, ci-l vom reţine, pe poziţia simetrică faţă de diagonala principala ($M[j][i]$).

4. Rezolvarea recursivă a relaţiei de recurenţă este ineficientă, datorită faptului că subproblemele se suprapun, deci o abordare recursivă ar conduce la rezolvarea aceleiaşi subprobleme de mai multe ori. Prin urmare vom rezolva relaţia de recurenţă în mod bottom-up: (determinăm parantezarea optimală a produselor de două matrice, apoi de 3 matrice, 4 matrice, etc).

```
long M[NMax][NMax];

void dinamic()
{
  int nr, i, j, k, kmin;
  long min, Infinit=1000000000;
  for(nr=2; nr<=n; nr++) //nr=cate matrice se inmultesc
  for (i=1; i<=n-nr+1; i++)
  {
    j=i+nr-1;
    //se inmultesc nr matrice, de la Ai la Aj
    for (k=i, min=Infinit; k<j; k++)
    //determin minimul si pozitia sa
    if (min>M[i][k]+M[k+1][j]+d[i-1]*d[k]*d[j])
    {
      min=M[i][k]+M[k+1][j]+d[i-1]*d[k]*d[j];
      kmin=k;
    }
```

```
      M[i][j]=min;
      M[j][i]=kmin;
    }
  }
```

Reconstituirea soluţiei optime se face foarte uşor în mod recursiv, utilizând informaţiile reţinute sub diagonala principală în matricea $M$:

```
void afisare(int i, int j)
{
  //afiseaza parantezarea optimala a produsului Aix...xAj
  if (i==M[j][i]) cout<<"A"<<i;
    else
    {
      cout<<"(";
      afisare(i,M[j][i]);
      cout<<")";
    }
  cout<<"x";
  if (j==M[j][i]+1) cout<<"A"<<j;
    else {cout<<"("; afisare(M[j][i]+1,j); cout<<")";}
}
```

### 15.2.2 Subşir crescător maximal

Fie un şir $A = (a_1, a_2, ..., a_n)$. Numim subşir al şirului $A$ o succesiune de elemente din $A$, în ordinea în care acestea apar în $A$:

$$a_{i_1}, a_{i_2}, ..., a_{i_k}, \text{ unde } 1 \leq i_1 < i_2 < ... < i_k \leq n.$$

Se cere determinarea unui subşir crescător al şirului $A$, de lungime maximă.

De exemplu, pentru

$$A = (8, 3, 6, 50, 10, 8, 100, 30, 60, 40, 80)$$

o soluţie poate fi

$$(3, 6, 10, 30, 60, 80).$$

*Rezolvare*

1. Fie $A_{i_1} = (a_{i_1} \leq a_{i_2} \leq ... \leq a_{i_k})$ cel mai lung subşir crescător al şirului $A$. Observăm că el coincide cu cel mai lung subşir crescător al şirului $(a_{i_1}, a_{i_1+1}, ..., a_n)$. Evident $A_{i_2} = (a_{i_2} \leq a_{i_3} \leq ... \leq a_{i_k})$ este cel mai lung subşir crescător al lui $(a_{i_2}, a_{i_2+1}, ..., a_n)$, etc. Prin urmare, o subproblemă a problemei iniţiale constă în determinarea celui mai lung subşir crescător care începe cu $a_i$, $i = \{1, .., n\}$.

Subproblemele nu sunt independente: pentru a determina cel mai lung subşir crescător care începe cu $a_i$, este necesar să determinăm cele mai lungi subşiruri crescătoare care încep cu $a_j$, $a_i \leq a_j$, $j = \{i+1, .., n\}$.

2. Pentru a reţine soluţiile subproblemelor vom considera doi vectori $l$ şi $poz$, fiecare cu $n$ componente, având semnificaţia:

$l[i]$ =lungimea celui mai lung subşir crescător care începe cu $a[i]$;

$poz[i]$ =poziţia elementului care urmează după $a[i]$ în cel mai lung subşir crescător care începe cu $a[i]$, dacă un astfel de element există, sau $-1$ dacă un astfel de element nu există.

3. Relaţia de recurenţă care caracterizează substructura optimală a problemei este:

$$l[n] = 1; poz[n] = -1;$$

$$l[i] = \max_{j=i+1,n} \{1 + l[j] | a[i] \leq a[j]\}$$

unde $poz[i]$ = indicele $j$ pentru care se obţine maximul $l[i]$.

Rezolvăm relaţia de recurenţă în mod bottom-up:

```
int i, j;
l[n]=1;
poz[n]=-1;
for (i=n-1; i>0; i--)
  for (l[i]=1, poz[i]=-1, j=i+1; j<=n; j++)
    if (a[i] <= a[j] && l[i]<1+l[j])
    {
      l[i]=1+l[j];
      poz[i]=j;
    }
```

Pentru a determina soluţia optimă a problemei, determinăm valoarea maximă din vectorul $l$, apoi afişăm soluţia, începând cu poziţia maximului şi utilizând informaţiile memorate în vectorul $poz$:

```
//determin maximul din vectorul l
int max=l[1], pozmax=1;
for (int i=2; i<=n; i++)
  if (max<l[i])
  {
    max=l[i];   pozmax=i;
  }
  cout<<"Lungimea celui mai lung subsir crescator: " <<max;
  cout<<"\nCel mai lung subsir:\n";
  for (i=pozmax; i!=-1; i=poz[i])   cout<<a[i]<<' ';
```

Secvenţele de program de mai sus sunt scrise în c/C++.

Programele următoare sunt scrise în Java:

```
import java.io.*;
class SubsirCrescatorNumere
{
  static int n;
  static int[] a;
  static int[] lg;
  static int[] poz;

  public static void main(String []args) throws IOException
  {
    int i,j;
    StreamTokenizer st=new StreamTokenizer(
           new BufferedReader(new FileReader("SubsirCrescatorNumere.in")));
    PrintWriter out = new PrintWriter (
           new BufferedWriter( new FileWriter("SubsirCrescatorNumere.out")));

    st.nextToken();n=(int)st.nval;
    a=new int[n+1];
    lg=new int[n+1];
    poz=new int[n+1];
    for(i=1;i<=n;i++) { st.nextToken(); a[i]=(int)st.nval; }
    int max,jmax;
    lg[n]=1;
    for(i=n-1;i>=1;i--)
    {
      max=0;
      jmax=0;
      for(j=i+1;j<=n;j++)
        if((a[i]<=a[j])&&(max<lg[j])) { max=lg[j]; jmax=j; }
      if(max!=0) { lg[i]=1+max; poz[i]=jmax; }
        else lg[i]=1;
    }
    max=0; jmax=0;
    for(j=1;j<=n;j++)
      if(max<lg[j]){ max=lg[j]; jmax=j; }
    out.println(max);
    int k;
    j=jmax;
    for(k=1;k<=max;k++) { out.print(a[j]+" "); j=poz[j]; }
    out.close();
  }//main
}//class
```

```
import java.io.*;
class SubsirCrescatorLitere
{

  public static void main(String[] args) throws IOException
  {
    int n,i,j,jmax,k,lmax=-1,pozmax=-1;
    int [] l,poz;
    String a;
    StreamTokenizer st=new StreamTokenizer(
            new BufferedReader(new FileReader("SubsirCrescatorLitere.in")));
    PrintWriter out=new PrintWriter(
            new BufferedWriter(new FileWriter("SubsirCrescatorLitere.out")));
    st.nextToken();
    a=st.sval;
    n=a.length();
    out.println(a+" "+n);
    l=new int[n];//l[i]=lg.celui mai lung subsir care incepe cu a[i]
    poz=new int[n];//poz[i]=pozitia elementului care urmeaza dupa a[i]
    for(i=0;i<n;i++) poz[i]=-1;
    l[n-1]=1;
    poz[n-1]=-1;
    for(i=n-2;i>=0;i--)
    {
      jmax=i;
      for(j=i+1;j<n;j++)
      if((a.charAt(i)<=a.charAt(j))&&(1+l[j]>1+l[jmax])) jmax=j;
      l[i]=1+l[jmax];
      poz[i]=jmax;
      if(l[i]>lmax) { lmax=l[i]; pozmax=i; }
    }
    out.print("Solutia ");
    k=pozmax;
    out.print("("+l[pozmax]+") : ");
    for(j=1;j<=lmax;j++)
    {
      out.print(a.charAt(k));
      k=poz[k];
    }
    out.close();
  } // main
}// class
```

### 15.2.3 Suma în triunghi

Să considerăm un triunghi format din $n$ linii ($1 < n \leq 100$), fiecare linie conţinând numere întregi din domeniul $[1, 99]$, ca în exemplul următor:

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

Tabelul 15.1: Triunghi de numere

Problema constă în scrierea unui program care să determine cea mai mare sumă de numere aflate pe un drum între numărul de pe prima linie şi un număr de pe ultima linie. Fiecare număr din acest drum este situat sub precedentul, la stânga sau la dreapta acestuia. (IOI, Suedia 1994)

*Rezolvare*

1. Vom reţine triunghiul într-o matrice pătratică $T$, de ordin $n$, sub diagonala principală. Subproblemele problemei date constau în determinarea sumei maxime care se poate obţine din numere aflate pe un drum între numarul $T[i][j]$, până la un număr de pe ultima linie, fiecare număr din acest drum fiind situat sub precedentul, la stânga sau la dreapta sa. Evident, subproblemele nu sunt independente: pentru a calcula suma maximă a numerelor de pe un drum de la $T[i][j]$ la ultima linie, trebuie să calculăm suma maximă a numerelor de pe un drum de la $T[i+1][j]$ la ultima linie şi suma maximă a numerelor de pe un drum de la $T[i+1][j+1]$ la ultima linie.

2. Pentru a reţine soluţiile subproblemelor, vom utiliza o matrice $S$, pătratică de ordin $n$, cu semnificaţia

$S[i][j]$ = suma maximă ce se poate obţine pe un drum de la $T[i][j]$ la un element de pe ultima linie, respectând condiţiile problemei.

Evident, soluţia problemei va fi $S[1][1]$.

3. Relaţia de recurenţa care caracterizează substructura optimală a problemei este:

$$S[n][i] = T[n][i], i = \{1, 2, ..., n\}$$

$$S[i][j] = T[i][j] + \max\{S[i+1][j], S[i+1][j+1]\}$$

4. Rezolvăm relaţia de recurenţă în mod bottom-up:

```
int i, j;
for (i=1; i<=n; i++) S[n][i]=T[n][i];
for (i=n-1; i>0; i--)
  for (j=1; j<=i; j++)
```

```
{
  S[i][j]=T[i][j]+S[i+1][j];
  if (S[i+1][j]<S[i+1][j+1])
    S[i][j]=T[i][j]+S[i+1][j+1]);
}
```

*Exerciţiu:* Afişaţi şi drumul în triunghi pentru care se obţine soluţia optimă.

### 15.2.4 Subşir comun maximal

Fie $X = (x_1, x_2, ..., x_n)$ şi $Y = (y_1, y_2, ..., y_m)$ două şiruri de $n$, respectiv $m$ numere întregi. Determinaţi un subşir comun de lungime maximă.

*Exemplu*

Pentru $X = (2, 5, 5, 6, 2, 8, 4, 0, 1, 3, 5, 8)$ şi $Y = (6, 2, 5, 6, 5, 5, 4, 3, 5, 8)$ o soluţie posibilă este: $Z = (2, 5, 5, 4, 3, 5, 8)$.

*Solutie*

1. Notăm cu $X_k = (x_1, x_2, ..., x_k)$ (prefixul lui $X$ de lungime $k$) şi cu $Y_h = (y_1, y_2, ..., y_h)$ prefixul lui $Y$ de lungime $h$. O subproblemă a problemei date constă în determinarea celui mai lung subşir comun al lui $X_k$, $Y_h$. Notam cu $LCS(X_k, Y_h)$ lungimea celui mai lung subşir comun al lui $X_k$, $Y_h$.

Utilizând aceste notaţii, problema cere determinarea $LCS(X_n, Y_m)$, precum şi un astfel de subşir.

*Observaţie*

1. Dacă $X_k = Y_h$ atunci

$$LCS(X_k, Y_h) = 1 + LCS(X_{k-1}, Y_{h-1}).$$

2. Dacă $Xk \neq Yh$ atunci

$$LCS(Xk, Yh) = \max(LCS(X_{k-1}, Y_h), LCS(X_k, Y_{h-1})).$$

Din observaţia precedentă deducem că subproblemele problemei date nu sunt independente şi că problema are substructură optimală.

2. Pentru a reţine soluţiile subproblemelor vom utiliza o matrice cu $n+1$ linii şi $m+1$ coloane, denumită $lcs$. Linia şi coloana 0 sunt utilizate pentru iniţializare cu 0, iar elementul $lcs[k][h]$ va fi lungimea celui mai lung subşir comun al şirurilor $X_k$ si $Y_h$.

3. Vom caracteriza substructura optimală a problemei prin următoarea relaţie de recurenţă:

$$lcs[k][0] = lcs[0][h] = 0, k = \{1, 2, .., n\}, h = \{1, 2, .., m\}$$

$$lcs[k][h] = 1 + lcs[k-1][h-1], \text{ dacă } x[k] = y[h]$$

$$\max lcs[k][h-1], lcs[k-1][h], \text{ dacă } x[k] <> y[h]$$

Rezolvăm relaţia de recurenţă în mod *bottom-up*:

```
for (int k=1; k<=n; k++)
  for (int h=1; h<=m; h++)
    if (x[k]==y[h])
      lcs[k][h]=1+lcs[k-1][h-1];
    else
      if (lcs[k-1][h]>lcs[k][h-1])
        lcs[k][h]=lcs[k-1][h];
      else
        lcs[k][h]=lcs[k][h-1];
```

Deoarece nu am utilizat o structură de date suplimentară cu ajutorul căreia să memorăm soluţia optimă, vom reconstitui soluţia optimă pe baza rezultatelor memorate în matricea *lcs*. Prin reconstituire vom obţine soluţia în ordine inversă, din acest motiv vom memora solutia într-un vector, pe care îl vom afişa de la sfârşit către început:

```
cout<<"Lungimea subsirului comun maximal: " <<lcs[n][m];
int d[100];
cout<<"\nCel mai lung subsir comun este: \n";
for (int i=0, k=n, h=m; lcs[k][h]; )
  if (x[k]==y[h])
  {
    d[i++]=x[k];
    k--;
    h--;
  }
  else
    if (lcs[k][h]==lcs[k-1][h])
      k--;
    else
      h--;
for (k=i-1;k>=0; k--)
  cout<< d[k] <<' ';
```

Secvenţele de cod de mai sus sunt scrise în C/C++. Programul următor este scris în Java şi determină toate soluţiile. Sunt determinate cea mai mică şi cea mai mare soluţie, în ordine lexicografică. De asemenea sunt afişate matricele auxiliare de lucru pentru a se putea urmării mai uşor modalitatea de determinare recursivă a tuturor soluţiilor.

```
import java.io.*; // SubsirComunMaximal
class scm
{
  static PrintWriter out;
  static int [][] a;
```

```
  static char [][] d;

  static String x,y;
  static char[] z,zmin,zmax;
  static int nsol=0,n,m;
  static final char  sus='|', stanga='-', diag='*';

  public static void main(String[] args) throws IOException
  {
    citire();
    n=x.length();
    m=y.length();
    out=new PrintWriter(new BufferedWriter( new FileWriter("scm.out")));

    int i,j;
    matrad();
    out.println(a[n][m]);
    afism(a);
    afism(d);

    z=new char[a[n][m]+1];
    zmin=new char[z.length];
    zmax=new char[z.length];

    System.out.println("O solutie oarecare");
    osol(n,m);// o solutie

    System.out.println("\nToate solutiile");
    toatesol(n,m,a[n][m]);// toate solutiile
    out.println(nsol);
    System.out.print("SOLmin = ");
    afisv(zmin);
    System.out.print("SOLmax = ");
    afisv(zmax);
    out.close();
    System.out.println("\n... Gata !!! ...");
  }

  static void citire() throws IOException
  {
    BufferedReader br=new BufferedReader(new FileReader("scm.in"));
    x=br.readLine();
    y=br.readLine();
  }
```

```
static void matrad()
{
  int i,j;
  a=new  int[n+1][m+1];
  d=new char[n+1][m+1];
  for(i=1;i<=n;i++)
    for(j=1;j<=m;j++)
      if(x.charAt(i-1)==y.charAt(j-1)) // x.charAt(i)==y.charAt(j) !
      {
        a[i][j]=1+a[i-1][j-1];
        d[i][j]=diag;
      }
      else
      {
        a[i][j]=max(a[i-1][j],a[i][j-1]);
        if(a[i-1][j]>a[i][j-1]) d[i][j]=sus; else d[i][j]=stanga;
      }
}

static void osol(int lin, int col)
{
  if((lin==0)||(col==0)) return;

  if(d[lin][col]==diag)
    osol(lin-1,col-1);
  else
    if(d[lin][col]==sus)
      osol(lin-1,col);
    else osol(lin,col-1);
  if(d[lin][col]==diag) System.out.print(x.charAt(lin-1));
}

static void toatesol(int lin, int col,int k)
{
  int i,j;
  if(k==0)
  {
    System.out.print(++nsol+" ");
    afisv(z);
    zminmax();
    return;
  }
  i=lin+1;
```

```
    while(a[i-1][col]==k)//merg in sus
    {
      i--;
      j=col+1;
      while(a[i][j-1]==k) j--;
      if( (a[i][j-1]==k-1)&&(a[i-1][j-1]==k-1)&&(a[i-1][j]==k-1))
      {
        z[k]=x.charAt(i-1);
        toatesol(i-1,j-1,k-1);
      }
    }//while
  }

  static void zminmax()
  {
    if(nsol==1)
    {
      copiez(z,zmin);
      copiez(z,zmax);
    }
    else
      if(compar(z,zmin)<0)
        copiez(z,zmin);
      else
        if(compar(z,zmax)>0) copiez(z,zmax);
  }

  static int compar(char[] z1, char[] z2)//-1=<; 0=identice; 1=>
  {
    int i=1;
    while(z1[i]==z2[i]) i++;// z1 si z2 au n componente 1..n
    if(i>n)
      return 0;
    else
      if(z1[i]<z2[i])
        return -1;
      else return 1;
  }

  static void copiez(char[] z1, char[] z2)
  {
    int i;
    for(i=1;i<z1.length;i++) z2[i]=z1[i];
  }
```

```
static int max(int a, int b)
{
  if(a>b) return a; else return b;
}

static void afism(int[][]a)// difera tipul parametrului !!!
{
  int n=a.length;
  int i,j,m;
  m=y.length();

  System.out.print("    ");
  for(j=0;j<m;j++) System.out.print(y.charAt(j)+" ");
  System.out.println();

  System.out.print("  ");
  for(j=0;j<=m;j++) System.out.print(a[0][j]+" ");
  System.out.println();

  for(i=1;i<n;i++)
  {
    System.out.print(x.charAt(i-1)+" ");
    m=a[i].length;

    for(j=0;j<m;j++) System.out.print(a[i][j]+" ");
    System.out.println();
  }
  System.out.println("\n");
}

static void afism(char[][]d)// difera tipul parametrului !!!
{
  int n=d.length;
  int i,j,m;
  m=y.length();

  System.out.print("    ");
  for(j=0;j<m;j++) System.out.print(y.charAt(j)+" ");
  System.out.println();

  System.out.print("  ");
  for(j=0;j<=m;j++) System.out.print(d[0][j]+" ");
  System.out.println();
```

```
    for(i=1;i<n;i++)
    {
      System.out.print(x.charAt(i-1)+" ");
      m=d[i].length;

      for(j=0;j<m;j++) System.out.print(d[i][j]+" ");
      System.out.println();
    }
    System.out.println("\n");
  }

  static void afisv(char[]v)
  {
    int i;
    for(i=1;i<=v.length-1;i++) System.out.print(v[i]);
    for(i=1;i<=v.length-1;i++) out.print(v[i]);
    System.out.println();
    out.println();
  }
}
```

Pe ecran apar următoarele rezultate:

```
    1 3 2 4 6 5 a c b d f e
  0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 1 1 1 1 1 1 1 1 1 1
2 0 1 1 2 2 2 2 2 2 2 2 2
3 0 1 2 2 2 2 2 2 2 2 2 2
4 0 1 2 2 3 3 3 3 3 3 3 3
5 0 1 2 2 3 3 4 4 4 4 4 4
6 0 1 2 2 3 4 4 4 4 4 4 4
a 0 1 2 2 3 4 4 5 5 5 5 5
b 0 1 2 2 3 4 4 5 5 6 6 6
c 0 1 2 2 3 4 4 5 6 6 6 6
d 0 1 2 2 3 4 4 5 6 6 7 7
e 0 1 2 2 3 4 4 5 6 6 7 7 8
f 0 1 2 2 3 4 4 5 6 6 7 8 8


    1 3 2 4 6 5 a c b d f e

1   * - - - - - - - - - - -
2   | - * - - - - - - - - -
3   | * - - - - - - - - - -
```

```
4   | | - * - - - - - - - - -
5   | | - | - * - - - - - - -
6   | | - | * - - - - - - - -
a   | | - | | - * - - - - - -
b   | | - | | - | - * - - - -
c   | | - | | - | * - - - - -
d   | | - | | - | | - * - - -
e   | | - | | - | | - | - *
f   | | - | | - | | - | * -


O solutie oarecare
1346acdf
Toate solutiile
1 1346acdf
2 1246acdf
3 1345acdf
4 1245acdf
5 1346abdf
6 1246abdf
7 1345abdf
8 1245abdf
9 1346acde
10 1246acde
11 1345acde
12 1245acde
13 1346abde
14 1246abde
15 1345abde
16 1245abde
SOLmin = 1245abde
SOLmax = 1346acdf

... Gata !!! ...
Press any key to continue...
```

# Capitolul 16

# Probleme

## 16.1 Probleme rezolvate

### 16.1.1 Poarta - OJI 2002

Se consideră harta universului ca fiind o matrice cu 250 de linii şi 250 de coloane. În fiecare celulă se găseşte o aşa numită poartă stelară, iar în anumite celule se găsesc echipaje ale porţii stelare.

La o deplasare, un echipaj se poate deplasa din locul în care se află în oricare alt loc în care se găseşte o a doua poartă, în cazul nostru în orice altă poziţie din matrice.

Nu se permite situarea simultană a mai mult de un echipaj într-o celulă. La un moment dat un singur echipaj se poate deplasa de la o poartă stelară la alta.

**Cerinţă**

Dându-se un număr $p$ ($1 < p < 5000$) de echipaje, pentru fiecare echipaj fiind precizate poziţia iniţială şi poziţia finală, determinaţi numărul minim de deplasări necesare pentru ca toate echipajele să ajungă din poziţia iniţială în cea finală.

**Datele de intrare**

Se citesc din fişierul text **poarta.in** în următorul format:

− pe prima linie numărul natural $p$ reprezentând numărul de echipaje,

− pe următoarele $p$ linii câte 4 numere naturale, primele două reprezentând coordonatele poziţiei iniţiale a unui echipaj (*linie coloană*), următoarele două reprezentând coordonatele poziţiei finale a aceluiaşi echipaj (*linie coloană*).

**Datele de ieşire**

Pe prima linie a fişierului text **poarta.out** se scrie un singur număr reprezentând numărul minim de deplasări necesar.

**Restricţii şi precizări**

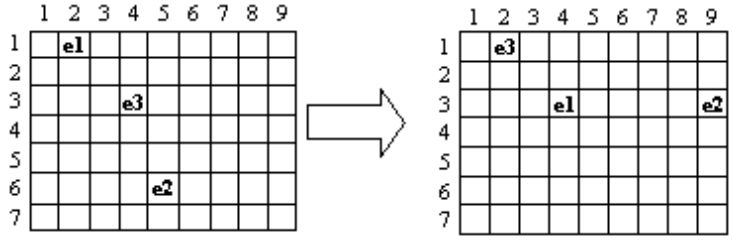
– coordonatele poziţiilor iniţiale şi finale ale echipajelor sunt numere naturale din intervalul $[1, 250]$;

– poziţiile iniţiale ale celor $p$ echipaje sunt distincte două câte două;

– poziţiile finale ale celor $p$ echipaje sunt distincte două câte două.

**Exemplu**

| poarta.in | poarta.out |
|---|---|
| 3<br>1 2 3 4<br>6 5 3 9<br>3 4 1 2 | 4 |




**Timp maxim de executare:** 1 secundă/test

```
import java.io.*; // NrMinDeplasari=p+NrCicluri-NrStationare
class poarta
{
  static int p,nmd,nc=0,ns=0;
  static int[] xi,yi,xf,yf;
  static boolean[] ea; // EsteAnalizat deja

  public static void main(String[] args) throws IOException
  {
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("poarta10.in")));

    st.nextToken(); p=(int)st.nval;
    xi=new int[p+1];
    yi=new int[p+1];
    xf=new int[p+1];
    yf=new int[p+1];
    ea=new boolean[p+1]; // EchipajAnalizat FALSE !!!
    int i;
    for(i=1;i<=p;i++)
    {
      st.nextToken(); xi[i]=(int)st.nval;
```

```
      st.nextToken(); yi[i]=(int)st.nval;
      st.nextToken(); xf[i]=(int)st.nval;
      st.nextToken(); yf[i]=(int)st.nval;
    }

    for(i=1;i<=p;i++)
    {
      if(ea[i]) continue;
      if((xf[i]==xi[i])&&(yf[i]==yi[i])) { ea[i]=true; ns++;}
        else if(ciclu(i)) nc++;
    }
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("poarta.out")));
    nmd=p+nc-ns;
    System.out.println(p+" "+nc+" "+ns+" "+nmd);
    out.print(nmd);
    out.close();
  }

  static boolean ciclu(int i)
  {
    int j=succesor(i);
    while((j!=-1)&&(j!=i))
    {
      ea[j]=true;
      j=succesor(j);
    }
    if(j==i) return true; else return false;
  }

  static int succesor(int j) // j --> k
  {
    int k;
    for(k=1;k<=p;k++)
      if((xf[j]==xi[k])&&(yf[j]==yi[k])) return k;
    return -1;
  }
}
```

### 16.1.2 Mouse - OJI 2002

Un experiment urmăreşte comportarea unui şoricel pus într-o cutie dreptunghiulară, împărţită în $m \times n$ cămăruţe egale de formă pătrată. Fiecare cămăruţă conţine o anumită cantitate de hrană.

Şoricelul trebuie să pornească din colţul $(1,1)$ al cutiei şi să ajungă în colţul opus, mâncând cât mai multă hrană. El poate trece dintr-o cameră în una alăturată (două camere sunt alăturate dacă au un perete comun), mănâncă toată hrana din cămăruţă atunci când intră şi nu intră niciodată într-o cameră fără hrană.

**Cerinţă**

Stabiliţi care este cantitatea maximă de hrană pe care o poate mânca şi traseul pe care îl poate urma pentru a culege această cantitate maximă.

**Datele de intrare**

Fişierul de intrare **mouse.in** conţine pe prima linie două numere $m$ şi $n$ reprezentând numărul de linii respectiv numărul de coloane ale cutiei, iar pe următoarele $m$ linii cele $m \times n$ numere reprezentând cantitatea de hrană existentă în fiecare cămăruţă, câte $n$ numere pe fiecare linie, separate prin spaţii. Toate valorile din fişier sunt numere naturale între 1 şi 100.

**Datele de ieşire**

În fişierul de ieşire **mouse.out** se vor scrie

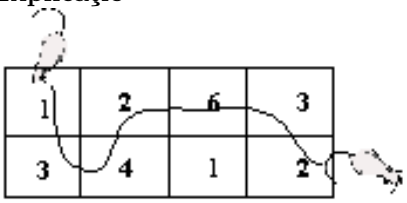
• pe prima linie două numere separate printr-un spaţiu: numărul de cămăruţe vizitate şi cantitatea de hrană maximă culeasă;

• pe următoarele linii un traseu posibil pentru cantitatea dată, sub formă de perechi de numere (linie coloană) începând cu 1 1 şi terminând cu $m$ $n$.

**Exemplu**

| mouse.in | mouse.out |
|---|---|
| 2 4<br>1 2 6 3<br>3 4 1 2 | 7 21<br>1 1<br>2 1<br>2 2<br>1 2<br>1 3<br>1 4<br>2 4 |

**Explicaţie**




**Timp maxim de executare:** 1 secundă/test

```
import java.io.*;
class Mouse
```

```
{
  static int m,n,imin,jmin,min,s;
  static int [][]a;
  static PrintWriter out;

  public static void main(String[] args) throws IOException
  {
    int i,j;
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("mouse4.in")));
    out=new PrintWriter(new BufferedWriter(new FileWriter("mouse.out")));

    st.nextToken();m=(int)st.nval;
    st.nextToken();n=(int)st.nval;
    a=new int[m+1][n+1];

    for(i=1;i<=m;i++)
      for(j=1;j<=n;j++) {st.nextToken(); a[i][j]=(int)st.nval;}

    s=0;
    for(i=1;i<=m;i++)
      for(j=1;j<=n;j++) s=s+a[i][j];

    if(m%2==1) mimpar();
      else if(n%2==1) nimpar();
             else mnpare();
    out.close();
  }//main

  static void mimpar()
  {
    int i,j;
    out.println(m*n+" "+s);
    i=1;
    while(i+1<m)
    {
      for(j=1;j<=n;j++) out.println(i+" "+j);
      i++;
      for(j=n;j>=1;j--) out.println(i+" "+j);
      i++;
    }
    for(j=1;j<=n;j++) out.println(m+" "+j);
  }
```

```
static void nimpar()
{
  int i,j;
  j=1;
  out.println(m*n+" "+s);
  while(j+1<n)
  {
    for(i=1;i<=m;i++) out.println(i+" "+j);
    j++;
    for(i=m;i>=1;i--) out.println(i+" "+j);
    j++;
  }
  for(i=1;i<=m;i++) out.println(i+" "+n);
}

static void mnpare()
{
  int i,j;
  imin=0;jmin=0;min=101;
  for(i=1;i<=m;i++)
    for(j=1;j<=n;j++)
      if((i+j)%2==1)
        if(a[i][j]<min) { min=a[i][j]; imin=i; jmin=j; }
  out.println((m*n-1)+" "+(s-a[imin][jmin]));

  j=1;
  while(j+1<jmin) // stanga
  {
    for(i=1;i<=m;i++) out.println(i+" "+j);
    j++;
    for(i=m;i>=1;i--) out.println(i+" "+j);
    j++;
  }

  i=1;
  while(i+1<imin) // sus
  {
    out.println(i+" " +j);
    out.println(i+" " +(j+1));
    out.println((i+1)+" " +(j+1));
    out.println((i+1)+" " +j);
    i=i+2;
  }
```

```
    out.println(i+" "+j); // patratel
    if((i==imin)&&(j+1==jmin)) out.println((i+1)+" " +j);
      else  out.println(i+" " +(j+1));
    out.println((i+1)+" " +(j+1));

    i=i+2;
    while (i<m) // jos
    {
      out.println(i+" " +(j+1));
      out.println(i+" " +j);
      out.println((i+1)+" " +j);
      out.println((i+1)+" " +(j+1));
      i=i+2;
    }

    j=j+2;
    while(j+1<=n) // dreapta
    {
      for(i=m;i>=1;i--) out.println(i+" "+j);
      j++;
      for(i=1;i<=m;i++) out.println(i+" "+j);
      j++;
    }
  }
}//class
```

### 16.1.3 Numere - OJI 2003

Gigel este un mare pasionat al cifrelor. Orice moment liber şi-l petrece jucându-se cu numere.

Jucându-se astfel, într-o zi a scris pe hârtie 10 numere distincte de câte două cifre şi a observat că printre acestea există două submulţimi disjuncte de sumă egală.

Desigur, Gigel a crezut că este o întâmplare şi a scris alte 10 numere distincte de câte două cifre şi spre surpriza lui, după un timp a găsit din nou două submulţimi disjuncte de sumă egală.

**Cerinţă**

Date 10 numere distincte de câte două cifre, determinaţi numărul de perechi de submulţimi **disjuncte** de sumă egală care se pot forma cu numere din cele date, precum şi una dintre aceste perechi pentru care suma numerelor din fiecare dintre cele două submulţimi este maximă.

**Date de intrare**

Fişierul de intrare **numere.in** conţine pe prima linie 10 numere naturale distincte separate prin câte un spaţiu $x_1\ x_2\ ...\ x_{10}$.

**Date de ieşire**

Fişierul de ieşire **numere.out** conţine trei linii. Pe prima linie se află numărul de perechi de submulţimi de sumă egală, precum şi suma maximă obţinută, separate printr-un spaţiu. Pe linia a doua se află elementele primei submulţimi, iar pe linia a treia se află elementele celei de a doua submulţimi, separate prin câte un spaţiu.

| | |
|---|---|
| **NrSol Smax** | **NrSol** - numărul de perechi; **Smax** - suma maximă |
| $x_1$ ... $x_k$ | elementele primei submulţimi |
| $y_1$ ... $y_p$ | elementele celei de a doua submulţimi |

**Restricţii şi precizări**

- $10 \leq x_i, y_i \leq 99$, pentru $1 \leq i \leq 10$.
- $1 \leq k, p \leq 9$.
- Ordinea submulţimilor în perechi nu contează.
- Perechea de submulţimi determinată nu este obligatoriu unică.

**Exemplu**

| **numere.in** | **numere.out** |
|---|---|
| 60 49 86 78 23 97 69 71 32 10 | 130 276<br>78 97 69 32<br>60 49 86 71 10 |

**Explicaţie:**

130 de soluţii; suma maximă este 276; s-au folosit 9 din cele 10 numere; prima submulţime are 4 elemente, a doua are 5 elemente.

**Timp maxim de executare:** 1 secundă/test

```
import java.io.*;
class Numere
{
static int[] x=new int[10];
static PrintWriter out;

public static void main(String[] args) throws IOException
{
int i, ia, ib, nsol=0, smax=-1, iamax=123,ibmax=123, sumaia=-1;
long t1,t2;
t1=System.currentTimeMillis();
StreamTokenizer st=new StreamTokenizer(
new BufferedReader(new FileReader("numere.in")));
out=new PrintWriter(new BufferedWriter(new FileWriter("numere.out")));
for(i=0;i<10;i++) {st.nextToken(); x[i]=(int)st.nval;}
for(ia=1;ia<=1022;ia++)
```

```
for(ib=ia+1;ib<=1022;ib++) // fara ordine intre A si B
// for(ib=1;ib<=1022;ib++) // cu   ordine intre A si B
// if(disjuncte(ia,ib))
if((ia&ib)==0) // 230,420 -> 80
{
//sumaia=suma(ia); // nu se castiga acum, dar ... !!!
if(suma(ia)==suma(ib)) // apel multiplu pentru suma(ia)
if(sumaia==suma(ib))
{
nsol++;
if(suma(ia)>smax)
//if(sumaia>smax)
{
smax=suma(ia);
//smax=sumaia;
iamax=ia;
ibmax=ib;
}
}
}
out.println(nsol+" "+smax);
afis(iamax);
afis(ibmax);
out.close();
t2=System.currentTimeMillis();
System.out.println(t2-t1);
}// main

static boolean disjuncte(int ia, int ib) // ia & ib == 0
{
int s=0;
while((ia!=0)&&(ib!=0))
{
 if((ia%2)*(ib%2)==1)  s++; // 420 milisecunde
// if((ia%2)*(ib%2)==1) { s++; break;} // 230 milisecunde !!!
ia/=2;
ib/=2;
}
if(s>0) return false; else return true;
}

static int suma(int i)
{
int s=0,k=0;
```

```
for(k=0;k<=9;k++) if( (i&(1<<k)) != 0 ) s+=x[k];
return s;
}

static void afis(int i)
{
int k=0;
while(i!=0)
{
if(i%2!=0) out.print(x[k]+" ");
k++;
i/=2;
}
out.println();
}
}// class
```

### 16.1.4 Expresie - OJI 2004

Se dă un şir de $n$ numere naturale nenule $x_1, x_2, ..., x_n$ şi un număr natural $m$.

**Cerinţă**

Să se verifice dacă valoarea expresiei $\sqrt[m]{x_1...x_n}$ exte un număr natural.

În caz afirmativ să se afişeze acest număr descompus în factori primi.

**Date de intrare**

În fişierul **exp.in** se află pe prima linie $m$, pe linia a doua $n$, iar pe linia a treia numerele $x_1, x_2, ..., x_n$ separate între ele prin câte un spaţiu.

**Date de ieşire**

În fişierul **exp.out** se va scrie pe prima linie cifra 0, dacă valoarea expresiei nu este un număr natural, respectiv 1 dacă este un număr natural. Dacă valoarea expresiei este un număr natural pe următoarele linii se vor scrie perechi de forma $p$ $e$ ($p$ este factor prim care apare în descompunere la puterea $e \geq 1$). Aceste perechi se vor scrie în ordine crescătoare după primul număr (adică $p$).

**Restricţii şi precizări**

- $n$ - număr natural nenul $< 5000$
- $x_i$ - număr natural nenul $< 30000$, $i \in 1, 2, ..., n$
- $m$ - poate fi una din cifrele 2, 3, 4

**Exemple**

| exp.in | exp.out | exp.in | exp.out |
|---|---|---|---|
| 2<br>4<br>32 81 100 19 | 0 | 2<br>4<br>32 81 100 18 | 1<br>2 4<br>3 3<br>5 1 |

**Timp maxim de execuţie:** 1 secundă/test

```
import java.io.*;
class expresie
{
  static int[] p=new int[30000];
  static int m,n;

  public static void main(String[] args) throws IOException
  {
    int i;
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("expresie.in")));
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;
    int[] x=new int[n+1];
    for(i=1;i<=n;i++) { st.nextToken(); x[i]=(int)st.nval; }
    for(i=1;i<=n;i++) descfact(x[i]);
    int ok=1;
    for (i=2;i<30000;i++)
    if (p[i]%m==0) p[i]=p[i]/m;
      else { ok=0; break; }
    PrintWriter out=new PrintWriter(
         new BufferedWriter(new FileWriter("expresie.out")));
    if(ok==0) out.println(0);
    else
    {
      out.println(1);
      for(i=2;i<30000;i++)
        if(p[i]!=0) out.println(i+" "+p[i]);
    }
    out.close();
  }

  static void descfact(int nr)
  {
    int d=2;
    while(nr%d==0)
    {
```

```
      p[d]++;
      nr=nr/d;
    }
    d=3;
    while(d<=nr)
    {
      while(nr%d==0) { p[d]++; nr=nr/d; }
      d=d+2;
    }
  }
}
```

### 16.1.5 Reactivi - OJI 2004

Într-un laborator de analize chimice se utilizează $N$ reactivi.

Se ştie că, pentru a evita accidentele sau deprecierea reactivilor, aceştia trebuie să fie stocaţi în condiţii de mediu speciale. Mai exact, pentru fiecare reactiv $x$, se precizează intervalul de temperatură $[min_x, max_x]$ în care trebuie să se încadreze temperatura de stocare a acestuia.

Reactivii vor fi plasaţi în frigidere.

Orice frigider are un dispozitiv cu ajutorul căruia putem stabili temperatura (constantă) care va fi în interiorul acelui frigider (exprimată într-un număr întreg de grade Celsius).

**Cerinţă**

Scrieţi un program care să determine numărul minim de frigidere necesare pentru stocarea reactivilor chimici.

**Date de intrare**

Fişierul de intrare **react.in** conţine:

– pe prima linie numărul natural $N$, care reprezintă numărul de reactivi;

– pe fiecare dintre următoarele $N$ linii se află $min\ max$ (două numere întregi separate printr-un spaţiu); numerele de pe linia $x + 1$ reprezintă temperatura minimă, respectiv temperatura maximă de stocare a reactivului $x$.

**Date de ieşire**

Fişierul de ieşire **react.out** va conţine o singură linie pe care este scris numărul minim de frigidere necesar.

**Restricţii şi precizări**

- $1 \leq N \leq 8000$
- $-100 \leq min_x \leq max_x \leq 100$ (numere întregi, reprezentând grade Celsius), pentru orice $x$ de la 1 la $N$
- un frigider poate conţine un număr nelimitat de reactivi

**Exemple**

| react.in | react.out | react.in | react.out | react.in | react.out |
|---|---|---|---|---|---|
| 3<br>-10 10<br>- 2 5<br>20 50 | 2 | 4<br>2 5<br>5 7<br>10 20<br>30 40 | 3 | 5<br>-10 10<br>10 12<br>-20 10<br>7 10<br>7 8 | 2 |

**Timp maxim de execuţie:** 1 secundă/test

```
import java.io.*;
class Reactivi
{
  static int n;        // n=nr reactivi
  static int ni=0;     // ni=nr interschimbari in quickSort
  static int nf=0;     // n=nr frigidere
  static int[] ngf;    // ng=nr grade in frigider
  static int[] x1,x2;  // limite inferioara/superioara

  public static void main(String[] args) throws IOException
  {
    int i,j;
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("Reactivi.in")));
    PrintWriter out=new PrintWriter(
         new BufferedWriter(new FileWriter("Reactivi.out")));
    st.nextToken(); n=(int)st.nval;
    x1=new int[n+1];
    x2=new int[n+1];
    ngf=new int[n+1];
    for(i=1;i<=n;i++)
    {
      st.nextToken(); x1[i]=(int)st.nval;
      st.nextToken(); x2[i]=(int)st.nval;
    }
    sol();
    out.println(nf);
    out.close();
  }// main

  static void sol()
  {
    int i;
    quickSort(1,n);
    i=1; nf=1; ngf[nf]=x2[i];
```

```
    i++;
    while(i<n)
    {
      while((i<=n)&&(x1[i]<=ngf[nf])) i++;
      if(i<n) ngf[++nf]=x2[i++];
    }
  }

  static void quickSort(int p, int u)
  {
    int i,j,aux;
    i=p; j=u;
    while(i<j)
    {
      while((i<j)&&((x2[i]<x2[j])||
           ((x2[i]==x2[j])&&(x1[i]>=x1[j])))) i++;
      if(i!=j)
      {
        aux=x1[i]; x1[i]=x1[j]; x1[j]=aux;
        aux=x2[i]; x2[i]=x2[j]; x2[j]=aux;
      }
      while((i<j)&&((x2[i]<x2[j])||
           ((x2[i]==x2[j])&&(x1[i]>=x1[j])))) j--;
      if(i!=j)
      {
        aux=x1[i]; x1[i]=x1[j]; x1[j]=aux;
        aux=x2[i]; x2[i]=x2[j]; x2[j]=aux;
      }
    }
    if(p<i-1) quickSort(p,i-1);
    if(i+1<u) quickSort(i+1,u);
  }
}
```

### 16.1.6 MaxD - OJI 2005

*Maria şi Adrian Niţă*

Fiind elev în clasa a IX-a, George, îşi propune să studieze capitolul divizibilitate cât mai bine. Ajungând la numărul de divizori asociat unui număr natural, constată că sunt numere într-un interval dat, cu acelaşi număr de divizori.

De exemplu, în intervalul $[1, 10]$, 6, 8 şi 10 au acelaşi număr de divizori, egal cu 4. De asemenea, 4 şi 9 au acelaşi număr de divizori, egal cu 3 etc.

**Cerinţă**

Scrieţi un program care pentru un interval dat determină care este cel mai mic număr din interval ce are număr maxim de divizori. Dacă sunt mai multe numere cu această proprietate se cere să se numere câte sunt.

**Date de intrare**

Fişierul de intrare **maxd.in** conţine pe prima linie două numere $a$ şi $b$ separate prin spaţiu ($a \leq b$) reprezentând extremităţile intervalului.

**Date de ieşire**

Fişierul de ieşire **maxd.out** va conţine pe prima linie trei numere separate prin câte un spaţiu *min nrdiv contor* cu semnificaţia:

*min* = cea mai mică valoare din interval care are număr maxim de divizori

*nrdiv* = numărul de divizori ai lui *min*

*contor* = câte numere din intervalul citit mai au acelaşi număr de divizori egal cu *nrdiv*

**Restricţii şi precizări**

- $1 \leq a \leq b \leq 1.000.000.000$
- $0 \leq b - a \leq 10.000$

**Punctaj**

Dacă aţi determinat corect *min*, obţineţi 50% din punctaj.

Dacă aţi determinat corect *nrdiv*, obţineţi 20% din punctaj

Dacă aţi determinat corect *contor*, obţineţi 30% din punctaj.

**Exemple**

| **maxd.in** | **maxd.out** | Explicaţie |
|---|---|---|
| 200 200 | 200 12 1 | 200 are 12 divizori iar in intervalul $[200, 200]$ există un singur număr cu această proprietate |

| **maxd.in** | **maxd.out** | Explicaţie |
|---|---|---|
| 2 10 | 6 4 3 | 6 este cel mai mic număr din interval care are maxim de divizori egal cu 4 şi sunt 3 astfel de numere 6, 8, 10 |

**Timp maxim de execuţie:** 1 sec/test

```
import java.io.*; //timp = 5600
class expresie
{
  static int[] x;
  static int a,b;

  public static void main(String[] args) throws IOException
  {
    int i;
    long t1,t2;
    t1=System.currentTimeMillis();
```

```
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("maxd.in")));
    st.nextToken(); a=(int)st.nval;
    st.nextToken(); b=(int)st.nval;
    x=new int[b-a+2];
    for(i=a;i<=b;i++) x[i-a+1]=descfact(i);
    int max=-1,n;
    int imax=-1;
    for(i=1;i<=b-a+1;i++)
      if(x[i]>max) { max=x[i]; imax=i; }
    int nrmax=0;
    for(i=1;i<=b-a+1;i++) if(x[i]==max) nrmax++;

    PrintWriter out=new PrintWriter(
         new BufferedWriter(new FileWriter("maxd.out")));
    out.println((imax+a-1)+" "+max+" "+nrmax);
    out.close();
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }

  static int descfact(int nr)
  {
    int d;
    int nrd;
    int p=1;

    d=2;
    nrd=0;
    while(nr%d==0) { nrd++; nr=nr/d; }
    p=p*(nrd+1);

    d=3;
    while(d*d<=nr)
    {
      nrd=0;
      while(nr%d==0) { nrd++; nr=nr/d; }
      p=p*(nrd+1);
      d=d+2;
    }
    if(nr>1) p*=2;
    return p;
  }
}
```

### 16.1.7 Fracţii - ONI 2001

*prof. Ovidiu Domşa, Alba Iulia*

O proprietate interesantă a fracţiilor ireductibile este că orice fracţie se poate obţine după următoarele reguli:

• pe primul nivel se află fracţia 1/1;

• pe al doilea nivel, în stânga fracţiei 1/1 de pe primul nivel, plasăm fracţia 1/2 iar în dreapta ei fracţia 2/1;

nivelul 1: 1/1
nivelul 2: 1/2 2/1

• pe fiecare nivel $k$ se plasează sub fiecare fracţie $i/j$ de pe nivelul de deasupra, fracţia $i/(i+j)$ în stânga, iar fracţia $(i+j)/j$ în dreapta.

nivelul 1: 1/1
nivelul 2: 1/2 2/1
nivelul 3: 1/3 3/2 2/3 3/1

**Cerinţă**

Dându-se o fracţie oarecare prin numărătorul şi numitorul său, determinaţi numărul nivelului pe care se află fracţia sau o fracţie echivalentă (având aceeaşi valoare) cu aceasta.

**Date de intrare**

Fişier de intrare: FRACTII.IN

• Linia 1: $N$ $M$ - două numere naturale nenule, separate printr-un spaţiu, reprezentând numărătorul şi numitorul unei fracţii.

**Date de ieşire**

Fişier de ieşire: FRACTII.OUT

• Linia 1: *niv* - număr natural nenul, reprezentând numărul nivelului care corespunde fracţiei.

**Restricţii**

$1 < N, M \leq 2.000.000.000$

**Exemple**

| FRACTII.IN | FRACTII.OUT | FRACTII.IN | FRACTII.OUT |
|---|---|---|---|
| 13 8 | 6 | 12 8 | 3 |

**Timp maxim de execuţie:** 1 secundă/test

```
import java.io.*;
class Fractii
{
public static void main(String[] args) throws IOException
{
int n,m,k,d;
```

```
StreamTokenizer st=new StreamTokenizer(new BufferedReader(
new FileReader("fractii.in")));
PrintWriter out=new PrintWriter(new BufferedWriter(
new FileWriter("fractii.out")));
st.nextToken();n=(int)st.nval;
st.nextToken();m=(int)st.nval;
k=0;
d=cmmdc(n,m);
n=n/d;
m=m/d;
while((n!=1)||(m!=1))
{
k++;
if(n>m) n=n-m; else m=m-n;
}
k++;
out.println(k);
out.close();
}

static int cmmdc(int a, int b)
{
int d,i,c,r;
if(a>b){d=a; i=b;} else {d=b; i=a;}
while(i!=0)
{
c=d/i;
r=d%i;
d=i;
i=r;
}
return d;
}
}
```

### 16.1.8 Competiţie dificilă - ONI 2001

*Angel Proorocu, Bucureşti*

La o competiţie au participat $N$ concurenţi. Fiecare dintre ei a primit un număr de concurs astfel încât să nu existe concurenţi cu acelaşi număr.

Numerele de concurs aparţin mulţimii $\{1, 2, ..., N\}$.

Din păcate, clasamentul final a fost pierdut, iar comisia îşi poate aduce aminte doar câteva relaţii între unii participanţi (de genul "participantul cu numărul 3 a ieşit înaintea celui cu numărul 5").

**Cerinţă**

Şeful comisiei are nevoie de un clasament final şi vă cere să-l ajutaţi determinând primul clasament în ordine lexicografică ce respectă relaţiile pe care şi le aminteşte comisia.

**Date de intrare**

Fişier de intrare: COMPET.IN

• Linia 1: $N$ $M$ - două numere naturale nenule, reprezentând numărul concurenţilor, respectiv numărul relaţiilor pe care şi le aminteşte comisia;

• Liniile 2..M+1: $i$ $j$ - pe fiecare din aceste $M$ linii se află câte două numere naturale nenule $i$ şi $j$, având semnificaţia: concurentul cu numărul de concurs $i$ a fost în clasament înaintea concurentului cu numărul de concurs $j$.

**Date de ieşire**

Fişier de ieşire: COMPET.OUT

• Linia 1: $nr_1$ $nr_2$ ... $nr_N$ - pe această linie se va scrie clasamentul sub forma unui şir de numere naturale nenule, separate prin câte un spaţiu, reprezentând numerele de concurs ale concurenţilor, în ordine de la primul clasat la ultimul.

**Restricţii şi precizări**

• $1 < N \leq 1000$

• se garantează corectitudinea datelor de intrare şi faptul că există totdeauna o soluţie.

**Exemple**

| COMPET.IN | COMPET.OUT | COMPET.IN | COMPET.OUT |
|---|---|---|---|
| 3 1<br>1 2 | 1 2 3 | 4 2<br>2 1<br>3 4 | 2 1 3 4 |

**Timp maxim de execuţie:** 1 secundă/test

```
import java.io.*;
class competitie
{
static int n,m;
static int[] a;
static int[] b;
static int[] c;
static int[] inainte;
static boolean[] finalizat;

public static void main(String[] args) throws IOException
{
int i,j,igasit=-1;
StreamTokenizer st=new StreamTokenizer(new BufferedReader(
```

```
new FileReader("competitie.in")));
PrintWriter out=new PrintWriter(new BufferedWriter(
new FileWriter("competitie.out")));
st.nextToken();n=(int)st.nval;
st.nextToken();m=(int)st.nval;
a=new int[m+1];
b=new int[n+1];
c=new int[n+1]; // solutia
inainte=new int[n+1];
finalizat=new boolean[n+1];
for(i=1;i<=m;i++)
{
st.nextToken();a[i]=(int)st.nval; // a[i] < b[i]
st.nextToken();b[i]=(int)st.nval;
}

for(i=1;i<=m;i++) inainte[b[i]]++;
for(j=1;j<=n;j++)
{
for(i=1;i<=n;i++)
if((!finalizat[i])&&(inainte[i]==0))
{
finalizat[i]=true;
c[j]=i;
igasit=i;
break;
}
for(i=1;i<=m;i++)
if(a[i]==igasit) inainte[b[i]]--;
}

for(i=1;i<=n;i++) out.print(c[i]+" ");
out.close();
}
}
```

### 16.1.9 Pentagon - ONI 2002

*lect. univ. Ovidiu Domşa, Alba Iulia*

În urma bombardamentelor din 11 septembrie 2001, clădirea Pentagonului a suferit daune la unul din pereţii clădirii. Imaginea codificată a peretelui avariat se reprezintă sub forma unei matrice cu $m$ linii şi $n$ coloane, ca în figura de mai jos:

1110000111 unde 1 reprezintă zid intact

1100001111 0 zid avariat
1000000011
1111101111
1110000111

Sumele alocate de Bin Laden pentru refacerea Pentagonului vor fi donate celor care vor ajuta americanii să refacă această clădire prin plasarea, pe verticală, a unor blocuri de înălţimi $k$, $k = 1, ..., m$, şi lăţime 1, în locurile avariate.

**Cerinţă:**

Pentru o structură dată a unui perete din clădirea Pentagonului, determinaţi numărul minim al blocurilor, de înălţimi $k = 1$, $k = 2$, ..., $k = m$, necesare refacerii clădirii.

**Date de intrare:**

Fişierul de intrare PENTAGON.IN conţine pe prima linie dimensiunile $m$ şi $n$ ale peretelui clădirii, iar pe următoarele $m$ linii câte o secvenţă de caractere 1 sau 0 de lungime $n$.

**Date de ieşire:**

Fişierul PENTAGON.OUT va conţine pe câte o linie, ordonate crescător după $k$, secvenţe:

$k$ $nr$

unde $k$ este înalţimea blocului,
iar $nr$ este numărul de blocuri de înălţime $k$, separate prin câte un spaţiu.

**Restricţii şi precizări**

- $1 \leq m, n \leq 200$
- nu se vor afişa blocurile de înălţime $k$, a căror număr este zero (0).

**Exemplu**

| PENTAGON.IN | PENTAGON.OUT |
|---|---|
| 5 10 | 1 7 |
| 1110000111 | 2 1 |
| 1100001111 | 3 2 |
| 1000000011 | 5 1 |
| 1111101111 | |
| 1110000111 | |

**Timp maxim de execuţie:** 1 secundă/test

```
import java.io.*;
class Pentagon
{
static int m,n;
static int [][]a;
static int []f;
```

```
public static void main(String[] args) throws IOException
{
int i,j,s;
StreamTokenizer st=new StreamTokenizer(
new BufferedReader(new FileReader("pentagon.in")));
PrintWriter out=new PrintWriter(
new BufferedWriter(new FileWriter("pentagon.out")));

st.nextToken();m=(int)st.nval;
st.nextToken();n=(int)st.nval;
a=new int[m+1][n+1];
f=new int[m+1];
for(i=1;i<=m;i++)
for(j=1;j<=n;j++)
{
st.nextToken();
a[i][j]=(int)st.nval;
}

    for(j=1;j<=n;j++)
    {
     i=1;
     while(i<=m)
     {
     s=0;
     while((i<=m)&&(a[i][j]==0)) // numararea 0-urilor
     {
     s++;
     i++;
     }
     if(s>0) f[s]++;
     while((i<=m)&&(a[i][j]==1)) i++; // sar peste 1
     }
    }

    for(i=1;i<=m;i++)
     if(f[i]!=0) out.println(i+" "+f[i]);
    out.close();
  }
}
```

### 16.1.10 Suma - ONI 2002

*Florin Gheţu, Bucureşti*

Fie şirul tuturor numerelor naturale de la 1 la un număr oarecare $N$.

Considerând asociate câte un semn (+ sau -) fiecărui număr şi adunând toate aceste numere cu semn se obţine o sumă $S$.

Problema constă în a determina pentru o sumă dată $S$, numărul minim $N$ pentru care, printr-o asociere de semne tuturor numerelor de la 1 la $N$, se poate obţine $S$.

**Cerinţă:**

Pentru un $S$ dat, găsiţi valoarea minimă $N$ şi asocierea de semne numerelor de la 1 la $N$ pentru a obţine $S$ în condiţiile problemei.

**Restricţii:**

• $0 < S \leq 100.000$.

**Date de intrare**

În fişierul SUMA.IN se va afla pe prima linie un întreg pozitiv $S$ reprezentând suma ce trebuie obţinută.

**Date de ieşire**

În fişierul SUMA.OUT se va scrie, pe prima linie numărul minim $N$ pentru care se poate obţine suma $S$ iar pe următoarele linii, până la sfârşitul fişierului, numerele care au semn negativ, câte unul pe linie.

Ordinea de afişare a numerelor nu are importanţă.

Celelalte numere care nu apar în fişier se consideră pozitive.

Dacă există mai multe soluţii se cere doar una.

**Exemplu:**

| SUMA.IN | SUMA.OUT |
|---|---|
| 12 | 7<br>1<br>7 |

Deci suma 12 se poate obţine din minimum 7 termeni astfel:

$$12 = -1 + 2 + 3 + 4 + 5 + 6 - 7.$$

Nu este singura posibilitate de asociere de semne termenilor de la 1 la 7.

**Timpul de execuţie:** 1 secundă/test

```
import java.io.*;
class Suma
{
public static void main(String[] args) throws IOException
{
int n=0,suma=0,s;
int np;
StreamTokenizer st=new StreamTokenizer(
new BufferedReader(new FileReader("suma.in")));
PrintWriter out=new PrintWriter(
```

```
new BufferedWriter(new FileWriter("suma.out")));
st.nextToken(); s=(int)st.nval;

while((suma<s)||((suma-s)%2==1))
{
n++;
suma=suma+n;
}
out.println(n);
np=(suma-s)/2;
if(np>0)
if(np<=n) out.println(np);
else out.println((np-n)+"\n"+n);
out.close();
  }
}
```

### 16.1.11 Maşina - ONI 2003

O ţară are $3 \leq N \leq 30000$ oraşe, numerotate de la 1 la $N$, dispuse pe un cerc.

PAM tocmai şi-a luat carnet de conducere şi vrea să viziteze toate oraşele ţării. Lui PAM îi este frică să conducă prin locuri aglomerate aşa că ea şi-a propus să meargă numai pe şoselele unde traficul este mai redus.

Există şosele de legătură între oricare două oraşe alăturate: între oraşul 1 şi oraşul 2, ... , între oraşul $i$ şi oraşul $i + 1$, iar oraşul $N$ este legat de oraşului 1.

Ca să nu se rătăcească, PAM şi-a propus să-şi aleagă un oraş de început şi să meargă pe şoselele respective în sens trigonometric până ajunge înapoi în oraşul de unde a plecat. Dacă PAM pleacă din oraşul $K$, atunci traseul ei va fi: $K, K + 1, ..., N, 1, 2, ..., K$.

Maşina lui PAM are un rezervor foarte mare (în care poate pune oricât de multă benzină). În fiecare oraş, PAM ia toată cantitatea de benzină existentă în oraş, iar parcurgerea fiecărei şosele necesită o anumită cantitate de benzină.

**Cerinţă**

Ştiind că PAM are, la începutul călătoriei, doar benzina existentă în oraşul de plecare, şi că, atunci când ajunge într-un oraş, ea va lua toată cantitatea de benzină disponibilă în acel oraş, să se găsească un oraş din care PAM îşi poate începe excursia astfel încât să nu rămână fără benzină.

Se consideră că PAM a rămas fără benzină dacă în momentul plecării dintr-un oraş, nu are suficientă benzină pentru a parcurge şoseaua care duce la oraşul următor. Dacă benzina îi ajunge la fix (adică la plecare are tot atâta benzină câtă îi trebuie) se consideră că PAM poate ajunge până în oraşul următor.

**Date de intrare**

Fişierul de intrare **masina.in** conţine

• pe prima linie numărul $N$,

• pe cea de-a doua linie se găsesc $N$ numere naturale $a[1]$, $a[2]$, ..., $a[N]$, separate prin câte un spaţiu, unde $a[i]$ reprezintă cantitatea de benzină disponibilă în oraşul $i$.

• linia a treia conţine un şir de $N$ numere naturale $b[1]$, $b[2]$, ..., $b[N]$, separate prin câte un spaţiu, unde $b[i]$ reprezintă cantitatea de benzină necesară străbaterii şoselei dintre oraşele $i$ şi $i+1$ (sau $N$ şi 1, dacă $i = N$).

**Date de ieşire**

Fişierul de ieşire **masina.out** va conţine un singur număr $s$ care reprezintă un oraş din care, dacă PAM îşi începe călătoria, poate completa turul ţării fără a face pana prostului.

**Restricţii şi precizări**

• Dacă există mai multe soluţii, se cere una singură.

• $0 \leq a[i] \leq 30000$

• $1 \leq b[i] \leq 30000$

**Exemplu**

| masina.in | masina.out |
|---|---|
| 6<br>0 3 2 5 10 5<br>7 8 3 2 1 4 | 4 |

**Timp maxim de execuţie:** 0.3 sec/test

```
import java.io.*;
class Masina
{
public static void main(String[] args) throws IOException
{
int i,rez;
int j,n;
StreamTokenizer st=new StreamTokenizer(
new BufferedReader(new FileReader("masina.in")));
PrintWriter out=new PrintWriter(
new BufferedWriter(new FileWriter("masina.out")));

st.nextToken();n=(int)st.nval;
int[] a=new int[n+1];
int[] b=new int[n+1];

for (j=1;j<=n;j++){ st.nextToken();a[j]=(int)st.nval;}
for (j=1;j<=n;j++){ st.nextToken();b[j]=(int)st.nval;}

for(i=1;i<=n;i++)
```

```
{
rez=a[i]-b[i];
j=i;
j++; if(j==n+1) j=1;
while((rez>=0)&&(j!=i))
{
rez=rez+a[j]-b[j];
j++;if(j==n+1) j=1;
}
if(rez>=0) {out.println(i); break;}
}
out.close();
}
}
```

### 16.1.12 Şir - ONI 2004

Gigel se distrează construind şiruri crescătoare de numere din mulţimea $\{1, 2, ..., n\}$. La un moment dat observă că unele şiruri, de cel puţin $k$ termeni ($k \geq 3$), au o proprietate mai aparte: diferenţa dintre doi termeni consecutivi este constantă. Iată câteva exemple de astfel de şiruri pentru $n \geq 21$:

2,3,4
1,5,9,13
7,10,13,16,19,21

**Cerinţă**

Dându-se numărul natural $n$ ajutaţi-l pe Gigel să numere câte astfel de şiruri poate să construiască.

**Date de intrare**

În fişierul de intrare **sir.in** se găseşte, pe prima linie, numărul $n$.

**Date de ieşire**

În fişierul de ieşire **sir.out** se va afişa, pe prima linie, numărul cerut urmat de caracterul sfârşit de linie.

**Restricţii:**

- $3 \leq n \leq 20000$
- $3 \leq k \leq n$

**Exemple:**

| sir.in | sir.out |
|---|---|
| 3 | 1 |
| 4 | 3 |
| 5 | 7 |

**Timp execuţie: 1 sec/test**

```
import  java.io.*;
class sir
{
  public static void main(String []args) throws IOException
  {
    int ns=0,n=19999,r,k,i;
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("sir.in")));
    PrintWriter out=new PrintWriter(new BufferedWriter(
         new FileWriter("sir.out")));
    st.nextToken(); n=(int)st.nval;
    ns=0;
    for(r=1;r<=(n-1)/2;r++)
      for(k=3;k<=(n-1+r)/r;k++)
        ns=ns+n-(k-1)*r;
    System.out.println(ns);
    out.println(ns);
    out.close();
  }
}
```

### 16.1.13 Triangulaţii - OJI 2002

O triangulaţie a unui poligon convex este o mulţime formată din diagonale ale poligonului care nu se intersectează în interiorul poligonului ci numai în vârfuri şi care împart toată suprafaţa poligonului în triunghiuri.

Fiind dat un poligon cu $n$ vârfuri notate $1, 2, ..., n$ să se genereze toate triangulaţiile distincte ale poligonului. Două triangulaţii sunt distincte dacă diferă prin cel puţin o diagonală.

**Date de intrare:** în fişierul text TRIANG.IN se află pe prima linie un singur număr natural reprezentând valoarea lui $n$ ($n \leq 11$).

**Date de ieşire:** în fişierul text TRIANG.OUT se vor scrie:

- pe prima linie, numărul de triangulaţii distincte;

- pe fiecare din următoarele linii câte o triangulaţie descrisă prin diagonalele ce o compun. O diagonală va fi precizată prin două numere reprezentând cele două vârfuri care o definesc; cele două numere ce definesc o diagonală se despart prin cel puţin un spaţiu, iar între perechile de numere ce reprezintă diagonalele dintr-o triangulaţie se va lăsa de asemenea minimum un spaţiu.

**Exemplu:**

TRIANG.IN

5

TRIANG.OUT

5

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

**Timp maxim de executare:**
7 secunde/test pe un calculator la 133 MHz.
3 secunde/test pe un calculator la peste 500 MHz.

```
class Triangulatie
{
  static int nv=5; // numar varfuri poligon
  static int n=nv-3; // numar diagonale in triangulatie
  static int m=nv*(nv-3)/2; // numarul tuturor diagonalelor
  static int[] x=new int[n+1];
  static int[] v1=new int[m+3],v2=new int[m+3];
  static int nsol=0;

  static void afisd()
  {
    int i;
    System.out.print(++nsol+" ==> ");
    for(i=1;i<=n;i++) System.out.print(v1[x[i]]+" "+v2[x[i]]+" ");
    System.out.println();
  }

  static void afisx()
  {
    int i;
    System.out.print(++nsol+" ==> ");
    for(i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }

  static void diagonale()
  {
    int i,j,k=0;
    i=1;
    for(j=3;j<=nv-1;j++) {v1[++k]=i; v2[k]=j;}
    for(i=2;i<=nv-2;i++)
      for(j=i+2;j<=nv;j++){v1[++k]=i; v2[k]=j;}
  }// diagonale()

  static boolean seIntersecteaza(int k, int i)
```

```
  {
    int j; // i si x[j] sunt diagonalele !!!
    for(j=1;j<=k-1;j++)
      if(((v1[x[j]]<v1[i])&&(v1[i]<v2[x[j]])&&(v2[x[j]]<v2[i]))||
         ((v1[i]<v1[x[j]])&&(v1[x[j]]<v2[i])&&(v2[i]<v2[x[j]]))
        ) return true;
    return false;
  }

  static void f(int k)
  {
    int i;
    for (i=x[k-1]+1; i<=m-n+k; i++)
    {
      if(seIntersecteaza(k,i)) continue;
      x[k]=i;
      if(k<n) f(k+1);
        // else afisx();
        else afisd();
    }
  }

  public static void main(String[] args)
  {
    diagonale();
    f(1);
  }
}// class
```

### 16.1.14 Spirala - OJI 2003

Se consideră un automat de criptare format dintr-un tablou cu $n$ linii şi $n$ coloane, tablou ce conţine toate numerele de la 1 la $n^2$ aşezate "şerpuit" pe linii, de la prima la ultima linie, pe liniile impare pornind de la stânga către dreapta, iar pe cele pare de la dreapta către stânga (ca în figură).

| | | | |
|---|---|---|---|
| $1 \rightarrow$ | $2 \rightarrow$ | $3 \rightarrow$ | $4 \downarrow$ |
| $8 \rightarrow$ | $7 \rightarrow$ | $6 \downarrow$ | $5 \downarrow$ |
| $\uparrow 9$ | 10 | $\leftarrow 11$ | $12 \downarrow$ |
| $\uparrow 16$ | $\leftarrow 15$ | $\leftarrow 14$ | $\leftarrow 13$ |

Numim "amestecare" operaţia de desfăşurare în spirală a valorilor din tablou în ordinea indicată de săgeţi şi de reaşezare a acestora în acelaşi tablou, "şerpuit" pe linii ca şi în cazul precedent.

| | | | |
|---|---|---|---|
| $1 \rightarrow$ | $2 \rightarrow$ | $3 \rightarrow$ | $4 \downarrow$ |
| $14 \downarrow$ | $\leftarrow 13$ | $\leftarrow 12$ | $\leftarrow 5$ |
| $15 \rightarrow$ | $16 \rightarrow$ | $9 \rightarrow$ | $8 \downarrow$ |
| 10 | $\leftarrow 11$ | $\leftarrow 6$ | $\leftarrow 7$ |

De exemplu, desfăşurarea tabloului conduce la şirul: 1 2 3 4 5 12 13 14 15 16 9 8 7 6 11 10, iar reaşezarea acestuia în tablou conduce la obţinerea unui nou tablou reprezentat în cea de-a doua figură.

După orice operaţie de amestecare se poate relua procedeul, efectuând o nouă amestecare. S-a observat un fapt interesant: că după un număr de amestecări, unele valori ajung din nou în poziţia iniţială (pe care o aveau în tabloul de pornire). De exemplu, după două amestecări, tabloul de 4x4 conţine 9 dintre elementele sale în exact aceeaşi poziţie în care se aflau iniţial (vezi elemente marcate din figură).

| | | | |
|---|---|---|---|
| **1** | **2** | **3** | **4** |
| 6 | **7** | 8 | **5** |
| 11 | **10** | 15 | 14 |
| **16** | 9 | 12 | **13** |

**Cerinţă**

Pentru $n$ şi $k$ citite, scrieţi un program care să determine numărul minim de amestecări ale unui tablou de $n$ linii necesar pentru a ajunge la un tablou cu exact $k$ elemente aflate din nou în poziţia iniţială.

**Date de intrare**

Fişierul de intrare **spirala.in** conţine pe prima linie cele două numere $n$ şi $k$ despărţite printr-un spaţiu.

**Date de ieşire**

Fişierul de ieşire **spirala.out** conţine o singură linie pe care se află numărul de amestecări determinat.

**Restricţii şi precizări**

$3 \leq N \leq 50$

Datele de intrare sunt alese astfel încât numărul minim de amestecări necesare să nu depăşească $2 * 10^9$

**Exemple**

| spirala.in | spirala.out | spirala.in | spirala.out |
|---|---|---|---|
| 4 9 | 2 | 6 36 | 330 |

**Timp maxim de executare/test:** 1 secundă

```
import java.io.*;
class spirala
{
  static int n,k;
  static int[] pr;
  static int[] p;

  public static void main(String[] args) throws IOException
  {
```

```
    int r=0,i;
    StreamTokenizer st=new StreamTokenizer(new BufferedReader(
          new FileReader("spirala.in")));
    PrintWriter out=new PrintWriter(new BufferedWriter(
         new FileWriter("spirala.out")));
    st.nextToken();n=(int)st.nval;
    st.nextToken();k=(int)st.nval;
    p=new int[n*n+1];
    pr=new int[n*n+1];
    constrp();
    r=1;
    for(i=1;i<=n*n;i++) pr[i]=p[i]; // p^1
    // afisv(p); afisv(pr); System.out.println("------------");
    while(!arekfixe()&&(r<21))
    {
      r++;
      pr=inmp(pr,p);
      // System.out.print("\nr="+r); afisv(pr);
    }
    out.println(r);
    out.close();
  }

  static void constrp()
  {
    int i,j,k,v,kp=0;
    int[][] a=new int[n+1][n+1];
    i=1; v=0;
    for(k=1;k<=n/2;k++)
    {
      for(j=1;j<=n;j++) a[i][j]=++v;
      for(j=n;j>=1;j--) a[i+1][j]=++v;
      i=i+2;
    }
    if(n%2==1)
      for(j=1;j<=n;j++) a[n][j]=++v;

    // afism(a);
    for(k=1;k<=n/2;k++) // dreptunghiul k
    {
      i=k;
      for(j=k;j<=n+1-k;j++) p[++kp]=a[i][j];
      j=n+1-k;
      for(i=k+1;i<=n-k;i++) p[++kp]=a[i][j];
```

```
      i=n+1-k;
      for(j=n+1-k;j>=k;j--) p[++kp]=a[i][j];
      j=k;
      for(i=n-k;i>=k+1;i--) p[++kp]=a[i][j];
    }
    // afisv(p);
  }

  static int[] inmp(int[]a,int[]b)
  {
    int k;
    int[] c=new int[n*n+1];
    for(k=1;k<=n*n;k++) c[k]=a[b[k]];
    return c;
  }

  static boolean arekfixe()
  {
    int i, s=0;
    for(i=1;i<=n*n;i++) if(pr[i]==i) s++;
    System.out.println("s="+s);
    if(s==k) return true; else return false;
  }

  static void afism(int[][] x)
  {
    int i,j;
    for(i=1;i<=n;i++)
    {
      System.out.println();
      for(j=1;j<=n;j++) System.out.print(x[i][j]+"\t");
    }
    System.out.println();
  }

  static void afisv(int[] x)
  {
    int i;
    System.out.println();
    for(i=1;i<=n*n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}
```

### 16.1.15 Partiţie - ONI 2003

Se defineşte o partiţie a unui număr natural $n$ ca fiind o scriere a lui $n$ sub forma:

$$n = n_1 + n_2 + ...nk, \qquad (k \geq 1)$$

unde $n_1, n_2, ..., n_k$ sunt numere naturale care verifică următoarea relaţie:

$$n_1 \geq n_2 \geq ... \geq n_i \geq ... \geq n_k \geq 1$$

**Cerinţă:** Fiind dat un număr natural $n$, să se determine câte partiţii ale lui se pot scrie, conform cerinţelor de mai sus, ştiind că oricare număr $n_i$ dintr-o partiţie trebuie să fie un număr impar.

**Date de intrare**

Fişierul **partitie.in** conţine pe prima linie numărul $n$

**Date de ieşire**

Fişierul **partitie.out** va conţine pe prima linie numărul de partiţii ale lui $n$ conform cerinţelor problemei.

**Exemplu:**

| **partitie.in** | **partitie.out** |
|---|---|
| 7 | 5 |

**Explicaţii:**

Cele cinci partiţii sunt:
1+1+1+1+1+1+1
1+1+1+1+3
1+1+5
1+3+3
7

**Restricţii:**

$1 \leq n \leq 160$

**Timp maxim de executare:** 3 secunde/test.

```
class PartitieNrImpare2 // n = suma de numere impare ;
{   // optimizat timp la jumatate !!!
  static int dim=0,nsol=0; // nsol = 38328320 2Gata!!!  timp = 4787
  static int[] x=new int[161];

  public static void main(String[] args)
  {
    long t1,t2;
    int n=160;
    int maxi=((n-1)/2)*2+1;
    t1=System.currentTimeMillis();
    f(n,maxi,1);
    t2=System.currentTimeMillis();
```

```
    System.out.println("nsol = "+nsol+" timp= "+(t2-t1));
  }

  static void f(int val, int maxip, int poz)
  {
    if(maxip==1)
    {
      nsol++;
      // dim=poz-1;
      // afis2(val);
      return;
    }
    if(val==0)
    {
      nsol++;
      // dim=poz-1;
      // afis1();
      return;
    }

    int maxi=((val-1)/2)*2+1;
    int maxiok=min(maxip,maxi);
    int i;
    for(i=maxiok;i>=3;i=i-2)
    {
      x[poz]=i;
      f(val-i,i,poz+1);
    }
    nsol++;
    // dim=poz-1;
    // afis2(val);
  }

  static void afis1()
  {
    int i;
    System.out.print("\n"+nsol+" : ");
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }

  static void afis2(int val)
  {
    int i;
    System.out.print("\n"+nsol+" : ");
```

```
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
    for(i=1;i<=val;i++) System.out.print("1 ");
  }

  static int max(int a,int b) { return (a>b)?a:b; }
  static int min(int a,int b) { return (a<b)?a:b; }
}// class
```

# Bibliografie


[1] Aho, A.; Hopcroft, J.; Ullman, J.D.; Data strutures and algorithms, Addison Wesley, 1983

[2] Aho, A.; Hopcroft, J.; Ullman, J.D.; The Random Access Machine, 1974

[3] Andonie R., Gârbacea I.; Algoritmi fundamentali, o perspectivă C++, Ed. Libris, 1995

[4] Apostol C., Roşca I. Gh., Roşca V., Ghilic-Micu B., Introducere în programare. Teorie şi aplicatii, Editura ... Bucureşti, 1993

[5] Atanasiu, A.; Concursuri de informatică. Editura Petrion, 1995

[6] Atanasiu, A.; Ordinul de complexitate al unui algoritm. Gazeta de Informatică nr.1/1993

[7] - Bell D., Perr M.: Java for Students, Second Edition, Prentice Hall, 1999

[8] Calude C.; Teoria algoritmilor, Ed. Universităţii Bucureşti, 1987

[9] Cerchez, E.; Informatică - Culegere de probleme pentru liceu, Ed. Polirom, Iaşi, 2002

[10] Cerchez, E., Şerban, M.; Informatică - manual pentru clasa a X-a., Ed. Polirom, Iaşi, 2000

[11] Cori, R.; Lévy, J.J.; Algorithmes et Programmation, Polycopié, version 1.6; `http://w3.edu.polytechnique.fr/informatique/`

[12] Cormen, T.H., Leiserson C.E., Rivest, R.L.; Introducere în Algoritmi, Ed Agora, 2000

[13] Cormen, T.H., Leiserson C.E., Rivest, R.L.; Pseudo-Code Language, 1994

[14] Cristea, V.; Giumale, C.; Kalisz, E.; Paunoiu, Al.; Limbajul C standard, Ed. Teora, Bucureşti, 1992

[15] Erickson J.; Combinatorial Algorithms; `http://www.uiuc.edu/~jeffe/`

[16] Flanagan, D.; Java in a Nutshell, O'Reilly, 1997.

[17] Flanagan, D.; Java examples in a Nutshell, O'Reilly, 1997.

[18] Giumale, C.; Introducere în Analiza Algoritmilor, Ed.Polirom, 2004

[19] Giumale C., Negreanu L., Călinoiu S.; Proiectarea şi analiza algoritmilor. Algoritmi de sortare, Ed. All, 1997

[20] Gosling, J.; Joy, B.; Steele, G.; The Java Language Specification, Addison Wesley, 1996.

[21] Knuth, D.E.; Arta programării calculatoarelor, vol. 1: Algoritmi fundamentali, Ed. Teora, 1999.

[22] Knuth, D.E.; Arta programarii calculatoarelor, vol. 2: Algoritmi seminumerici, Ed. Teora, 2000.

[23] Knuth, D.E.; Arta programarii calculatoarelor, vol. 3: Sortare şi căutare, Ed. Teora, 2001.

[24] Lambert,K. A., Osborne,M.; Java. A Framework for Programming and Problem Solving, PWS Publishing, 1999

[25] Livovschi, L.; Georgescu H.; Analiza şi sinteza algoritmilor. Ed. Enciclopedică, Bucureşti, 1986.

[26] Niemeyer, P.; Peck J.; Exploring Java, O'Reilly, 1997.

[27] Odăgescu, I.; Smeureanu, I.; Ştefănescu, I.; Programarea avansată a calculatoarelor personale, Ed. Militară, Bucureşti 1993

[28] Odăgescu, I.; Metode şi tehnici de programare, Ed. Computer Lobris Agora, Cluj, 1998

[29] Popescu Anastasiu, D.; Puncte de articulaţie şi punţi în grafuri, Gazeta de Informatică nr. 5/1993

[30] Rotariu E.; Limbajul Java, Computer Press Agora, Tg. Mures, 1996

[31] Tomescu, I.; Probleme de combinatorică şi teoria grafurilor, Editura Didactică şi Pedagogică, Bucureşti, 1981

[32] Tomescu, I.; Leu, A.; Matematică aplicată în tehnica de calcul, Editura Didactică şi Pedagogică, Bucureşti, 1982

[33] Vaduva, C.M.; Programarea in JAVA. Microinformatica, 1999

[34] iţnescu, R.;; Viţnescu, V.; Programare dinamică - teorie şi aplicaţii; GInfo nr. 15/4 2005

[35] Vlada, M.; Conceptul de algoritm - abordare modernă, GInfo, 13/2,3 2003

[36] Vlada, M.; Grafuri neorientate şi aplicaţii. Gazeta de Informatică, 1993

[37] Weis, M.A.; Data structures and Algorithm Analysis, Ed. The Benjamin/Cummings Publishing Company. Inc., Redwoods City, California, 1995.

[38] Winston, P.H., Narasimhan, S.: On to JAVA, Addison-Wesley, 1996

[39] Wirth N., Algorithms + Data Structures = Programs, Prentice Hall, Inc 1976

[40] *** - Gazeta de Informatică, Editura Libris, 1991-2005