ALGORITMI ŞI STRUCTURI DE DATE

Note de curs

(uz intern - draft v2.3)

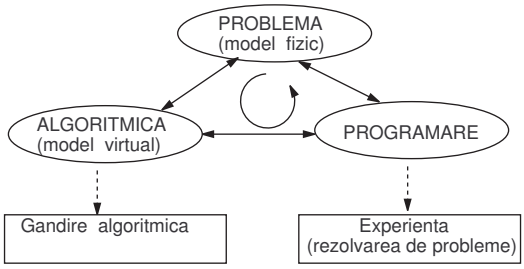
Prefaţă

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
```

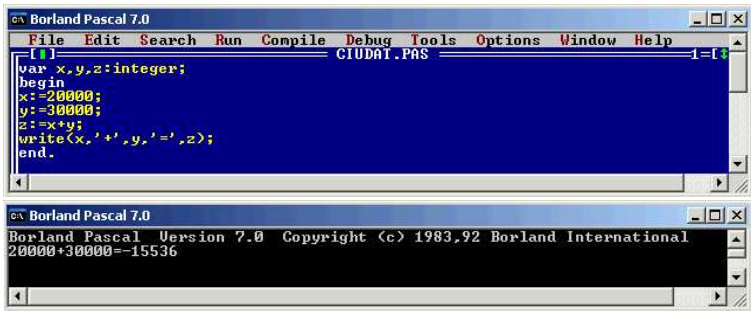


Figura 1.1: Un program ciudat în Pascal

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
```

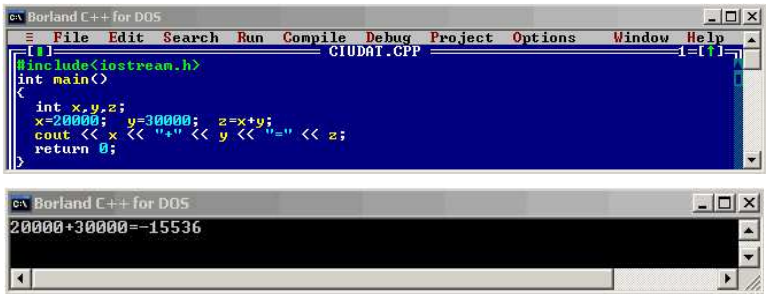


Figura 1.2: Un program ciudat în C++

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
```

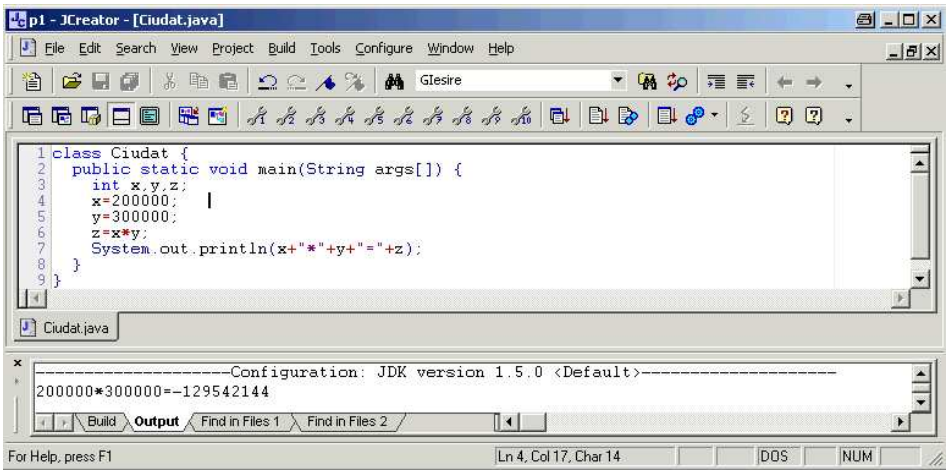


Figura 1.3: Un program ciudat în Java

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 6×10^4, 4×10^9 şi respectiv 2×10^{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

1101111110000100011101011000000000000

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):

111110000100011101011000000000000 (cod complementar)
100001111011100010100111111111111 (cod invers)
100001111011100010101000000000000 (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.

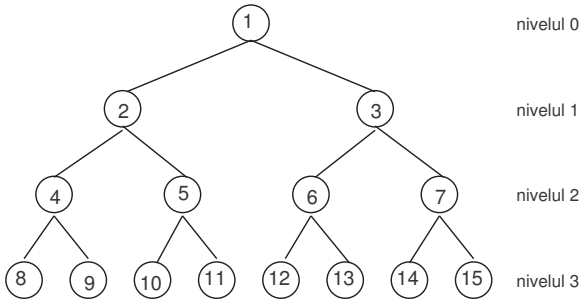


Figura 2.2: Arbore binar plin

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).

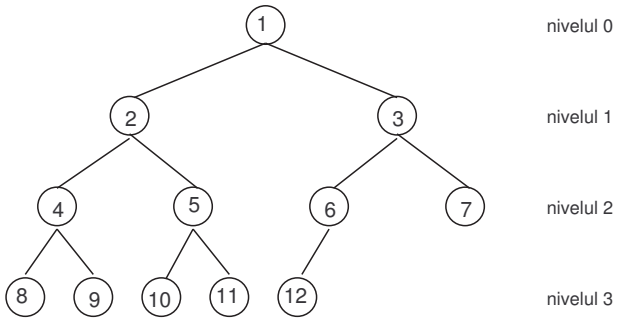


Figura 2.3: Arbore binar complet

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.

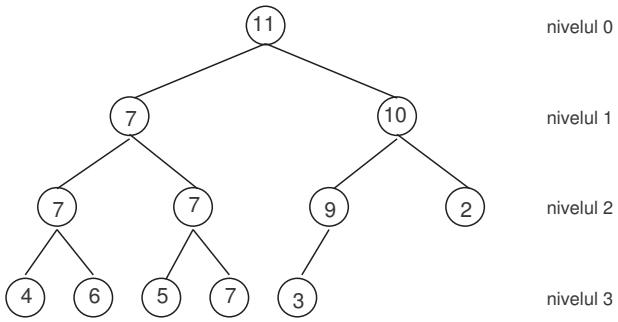


Figura 2.4: Max-heap

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 ε, limita şirului

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

Rezolvare. Calculul aproximativ (cu precizia ε) 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:

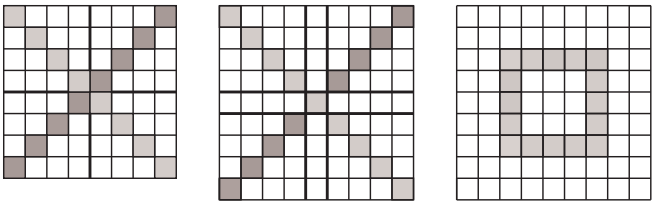


Figura 3.1: Zone în matrice pătratică

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 ξ 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 74009586436825301627l188139587625
6 : 37 170 11218727630940119874547782377l2816687
7 : 39 172 355716059292752464797065038013137686280
8 : 41 163 350417311326100987713326915256638659028 50
9 : 43 189 13850225097959561846019070897007305096801 95
10 : 44 205 261547165158628812920127773965779937817270 11
11 : 45 177 26723575409002161865117527704693137105019478 0
12 : 46 205 161933094493627977915438174581642803644128641 0
13 : 46 232 623890127627578481149286179482673756388928823 0
14 : 47 205 161328092700664943761253229880356919811584909 30
15 : 47 162 29226457001965139089793853213126510270024300000
16 : 47 216 384008253654955448238478079885360718157800509 40
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 ϕ 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

$$En = E_2E_{n-1} + E_3E_{n-2} + ... + E_{n-1}E_2, E_1 = E_2 = 1.$$

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 $O(f) = O(g) \Longleftrightarrow f \in O(g)$ *şi* $g \in O(f)$

Proprietatea 2 $O(f) \subset O(g) \Longleftrightarrow f \in O(g)$ *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 Ω.

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 Θ 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:*

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

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

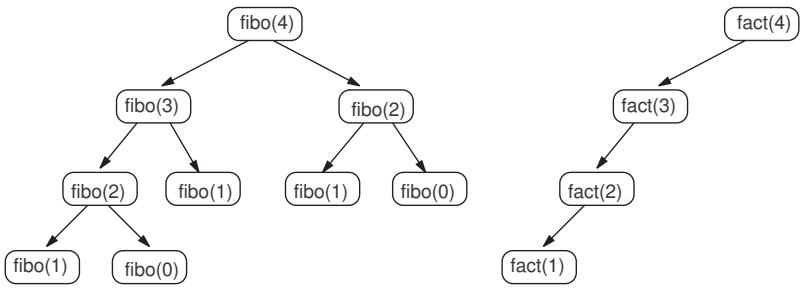
Pentru calculul termenilor şirului lui Fibonacci, a transcriere literală a formulei este următoarea:

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

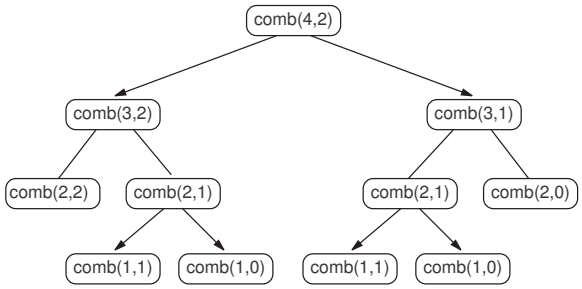
`fib` este o funcţie care utilizează propriul nume în definiţia proprie. De asemenea, dacă argumentul n este mai mic decât 1 returnează valoarea 1 iar în caz contrar returnează $fib(n-1) + fib(n-2)$.

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 cĭu 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!

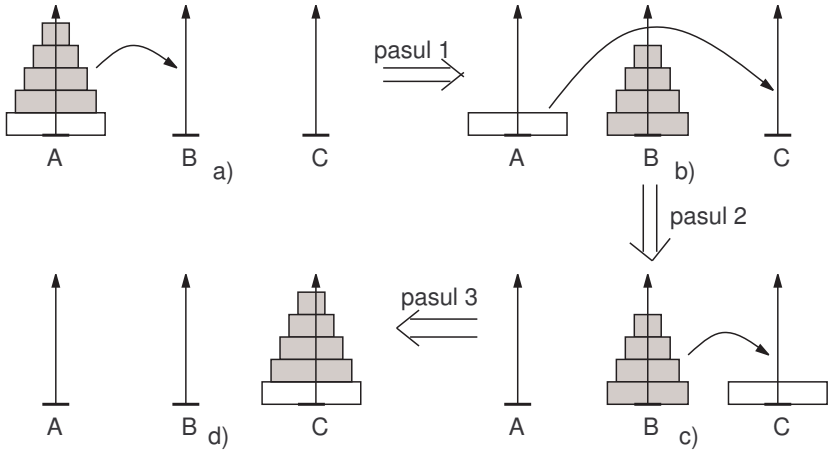
5.2 Proceduri recursive

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 tijăj, 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:

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

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

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

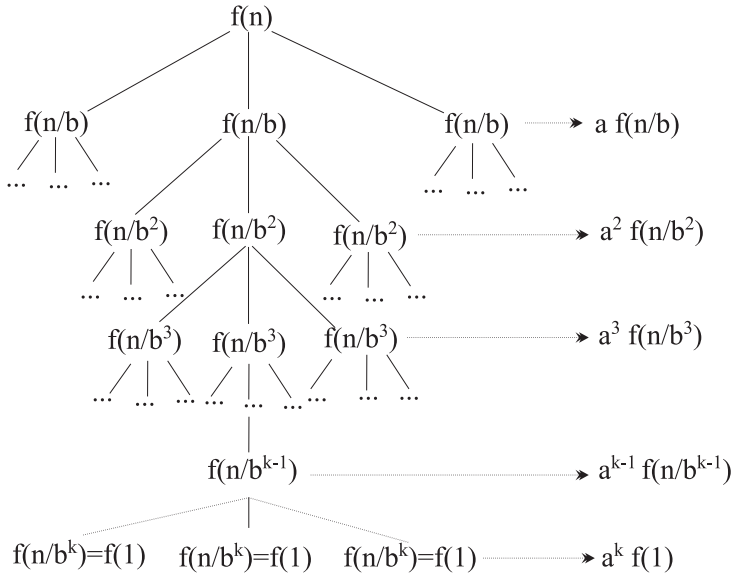
$$T(n) = aT(n/b) + f(n) \tag{6.2.6}$$

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 aceeaşi 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 α 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 α, 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:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3), n \geq 3, \text{ cu } T(0) = 0, T(1) = 1, T(2) = 2.$$

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:

$$\begin{array}{llllllll} x_3 & = & 0, & y_3 & & = & 1; \\ x_2 & = & 1, & y_2 & = 0-1*2 & = & -2; \\ x_1 & = & -2, & y_1 & = 1+2*1 & = & 3; \\ x_0 & = & 3, & y_1 & = -2-3*3 & = & -11. \end{array}$$

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

$$a(X) = a_m X^m + ... + a_1 X + a_0 \text{ şi } b(X) = b_n X^n + ... + b_1 X + b_0.$$

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 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;
  }
}
```

8.1.3 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:

$$|A_1 \cup A_2 \cup ... \cup A_n| = \sum_{i=1}^{n} |A_i| - \sum_{1 \leq i < j \leq n} |A_i \cap A_j| + ... + (-1)^{n-1} |\bigcap_{i=1}^{n} A_i|.$$

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:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ pentru } n \geq 2. \tag{8.3.1}$$

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:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \tag{8.3.2}$$

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n \tag{8.3.3}$$

$$F_{n+m} = F_mF_{n+1} + F_{m-1}F_n \tag{8.3.4}$$

$$F_{nk} = \text{ multiplu de } F_k \tag{8.3.5}$$

$$\tag{8.3.6}$$

şi

$$F_2 + F_4 + ... + F_{2n} = F_{2n+1} - 1 \tag{8.3.7}$$

$$F_1 + F_3 + ... + F_{2n-1} = F_{2n} \tag{8.3.8}$$

$$F_1^2 + F_2^2 + ... + F_n^2 = F_nF_{n+1} \tag{8.3.9}$$

$$F_1F_2 + F_2F_3 + ... + F_{2n-1}F_{2n} = F_{2n}^2 \tag{8.3.10}$$

$$F_1F_2 + F_2F_3 + ... + F_{2n}F_{2n+1} = F_{2n+1}^2 - 1 \tag{8.3.11}$$

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 Descompunerea în factori primi

8.4.1 Funcţia 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();
  }
}
```

8.4.2 Numărul divizorilor

Fie

$$n = f_1^{e_1} \cdot f_2^{e_2} \cdot ... \cdot f_k^{e_k}$$

descompunerea lui n în factori primi şi $ndiv(n)$ numărul divizorilor lui n. Atunci

$$ndiv(n) = (1 + e_1) \cdot (1 + e_2) \cdot ... \cdot (1 + e_k).$$

8.4.3 Suma divizorilor

Fie

$$n = f_1^{e_1} \cdot f_2^{e_2} \cdot ... \cdot f_k^{e_k}$$

descompunerea lui n în factori primi şi $sdiv(n)$ suma divizorilor lui n. Atunci

$$sdiv(n) = \sum_{d|n} d = \frac{f_1^{(1+e_1)} - 1}{f_1 - 1} \cdot \frac{f_2^{(1+e_2)} - 1}{f_2 - 1} \cdot ... \cdot \frac{f_k^{(1+e_k)} - 1}{f_k - 1}.$$

Demonstraţie:

Fie $n = ab$ cu $a \neq b$ şi $cmmdc(a, b) = 1$. Pentru orice divizor d al lui n, $d = a_i b_j$, unde a_i este divizor al lui a iar b_j este divizor al lui b. Divizorii lui a sunt: $1, a_1, a_2, ..., a$. Divizorii lui b sunt: $1, b_1, b_2, ..., b$.

Sumele divizorilor lui a şi b sunt:

$$sdiv(a) = 1 + a_1 + a_2 + ... + a$$

$$sdiv(b) = 1 + b_1 + b_2 + ... + b.$$

Dar

$$sdiv(ab) = \sum_{d|ab} d = \sum_{i,j} a_i b_j = (\sum_i a_i) \cdot (\sum_j b_j) = sdiv(a) \cdot sdiv(b)$$

De aici rezultă că:

$$sdiv(f_1^{e_1} \cdot f_2^{e_2} \cdot ... \cdot f_k^{e_k}) = sdiv(f_1^{e_1}) \cdot sdiv(f_2^{e_2}) \cdot ... \cdot sdiv(f_k^{e_k})$$

şi mai departe rezultă relaţia dată iniţial!

8.5 Partiţia numerelor

8.5.1 Partiţia lui n în exact k termeni

Fie $P(n, k)$ numărul modalităţilor de a descompune numărul natural n ca sumă de **exact** k termeni nenuli, considerând două descompuneri ca fiind distincte dacă diferă prin cel puţin un termen (deci, fără a ţine cont de ordinea termenilor).

Atunci

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

unde

$$P(i, 1) = P(i, i) = 1 \text{ pentru } (\forall) i \geq 1$$

şi

$$n = a_1 + a_2 + ... + a_k; \quad a_1 \geq a_2 \geq ... \geq a_k \geq 1.$$

Demonstraţie:

Ultimul termen a_k poate fi 1 sau ≥ 2.

Pentru $a_k = 1$ avem $P(n - 1, k - 1)$ posibilităţi de alegere a factorilor a_1, a_2, ..., a_{k-1} astfel încât $n - 1 = a_1 + a_2 + ... + a_{k-1}$ şi $a_1 \geq a_2 \geq ... \geq a_{k-1} \geq 1$

Pentru $a_k \geq 2$ putem să scădem 1 din toţi factorii descompunerii lui n şi obţinem $n - k = a'_1 + a'_2 + ... + a'_k$ unde $a'_1 \geq a'_2 \geq ... \geq a'_k \geq 1$, deci numărul descompunerilor lui n cu $a_k \geq 2$ este $P(n - k, k)$.

Obţinem relaţia

$$P(n, k) = P(n - 1, k - 1) + P(n - k, k)$$

care poate fi transformată astfel:

$$\begin{aligned}
P(n,k) &= P(n-1,k-1) + P(n-k,k)\\
P(n-1,k-1) &= P(n-2,k-2) + P(n-k,k-1)\\
P(n-2,k-2) &= P(n-3,k-3) + P(n-k,k-2)\\
&\ldots\\
P(n-k+3,3) &= P(n-k+2,2) + P(n-k,3)\\
P(n-k+2,2) &= P(n-k+1,1) + P(n-k,2)
\end{aligned}$$

Prin reducere şi ţinând cont că $P(n-k+1,1) = 1 = P(n-k,1)$ obţinem relaţia dată la început.

8.5.2 Partiţia lui n în cel mult k termeni

Fie $A(n,k)$ numărul modalităţilor de a descompune numărul natural n ca sumă de **cel mult** k termeni nenuli, considerând două descompuneri ca fiind distincte dacă diferă prin cel puţin un termen (deci, fără a ţine cont de ordinea termenilor).

Atunci

$$A(n,k) = A(n,k-1) + A(n-k,k)$$

Evident $A(i,j) = A(i,i)$ pentru orice $j > i$ iar $A(n,n)$ reprezintă numărul partiţiilor lui n.

(1) Numărul partiţiilor în cel mult k factori este egal cu numărul partiţiilor în **exact** k factori plus numărul partiţiilor în **cel mult** $k-1$ termeni.

(2) Dată fiind o partiţie a lui n în exact k termeni nenuli, putem scădea 1 din fiecare termeni, obţinând o partiţie a lui $n-k$ în **cel mult** k termeni nenuli. Astfel, există o corespondenţă bijectivă între partiţiile lui n în **exact** k termeni şi partiţiile lui $n-k$ în **cel mult** k factori.

8.5.3 Partiţii multiplicative

Fie $A(n,k)$ numărul modalităţilor de a descompune numărul natural n ca sumă de **cel mult** k termeni nenuli, considerând două descompuneri ca fiind distincte dacă diferă prin cel puţin un termen (deci, fără a ţine cont de ordinea termenilor).

8.6 Partiţia mulţimilor

Fie $S(n,k)$ numărul modalităţilor de a partiţiona o mulţime A cu n elemente în k submulţimi nevide, considerând două partiţii ca fiind distincte dacă diferă prin cel puţin o submulţime (deci, fără a ţine cont de ordinea submulţimilor).

Atunci

$$S(n,k) = S(n-1,k-1) + k \cdot S(n-1,k)$$

unde

$$S(i,1) = S(i,i) = 1 \text{ pentru } (\forall) i \geq 1$$

şi

$$A = A_1 \cup A_2 \cup ... \cup A_k; \quad A_i \cap A_j = \emptyset \text{ pentru } i \neq j.$$

Demonstraţie:

Fie

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

o mulţime cu n elemente şi

$$A_1, A_2, ..., A_k$$

o partiţie oarecare.

Elementul a_n poate fi

(1) într-o submulţime cu un singur element (chiar el!), sau

(2) într-o submulţime cu cel puţin 2 elemente (printre care se găşeşte şi el!).

Numărul partiţiilor de tipul (1) este $S(n-1,k-1)$ (fără elementul a_n rămân $n-1$ elemente şi $k-1$ submulţimi în partiţie).

Numărul partiţiilor de tipul (2) este $k \cdot S(n-1,k)$ (eliminând elementul a_n din submulţimea în care se află, acea submulţime rămâne nevidă!). Se pare că nu este prea clar (sau evident!) de ce apare totuşi factorul k în expresia $k \cdot S(n-1,k)$! Să privim puţin altfel lucrurile: considerăm toate partiţiile mulţimii $\{a_1, a_2, ..., a_{n-1}\}$ care au k submulţimi; numărul acestora este $S(n-1,k)$; introducerea elementului a_n în aceste partiţii se poate face în oricare din cele k submulţimi; deci $k \cdot S(n-1,k)$ reprezintă numărul partiţiilor din cazul (2).

8.7 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 afă 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 înainte, $\log_2 N$ 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.

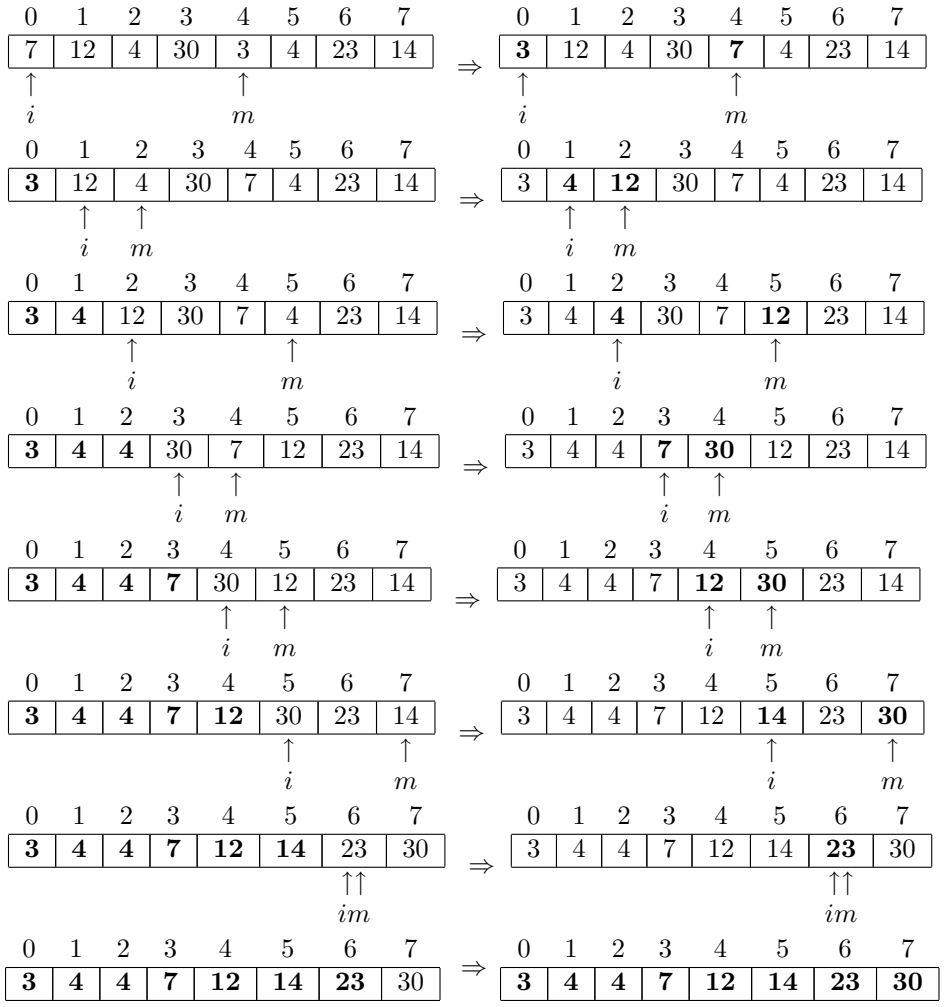
De această dată vom prezenta programul complet.

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:



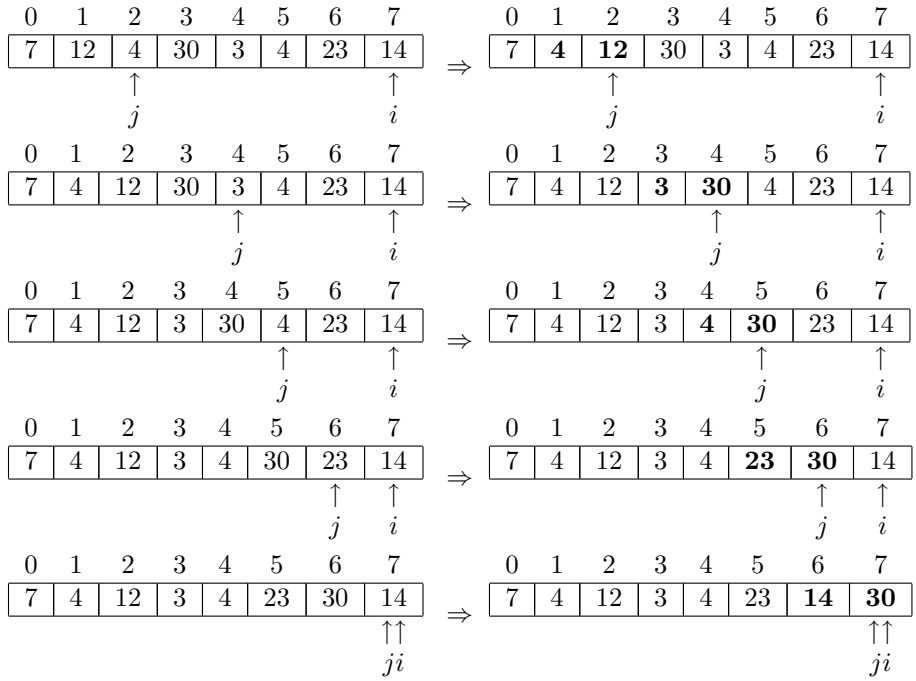
Este uşor de calculat numărul de operaţii necesare. La fiecare iteraţie, se pleacă de la elementul a_i şi se compară succesiv cu a_{i+1}, a_{i+2}, ..., a_N. Se fac deci $N - i$ comparaţii. Se începe cu $i = 1$ şi se termină cu $i = N - 1$. Deci, se fac $(N-1) + (N-2) + ... + 2 + 1 = N(N-1)/2$ comparaţii 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).

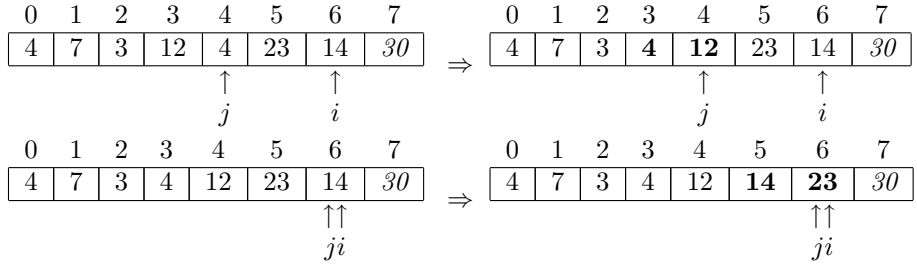
Procedura corespunzătoare utilizează un indice i care marchează sfârşitul prefixului în sortare, şi un indice j care permite deplasarea către marginea i.

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).

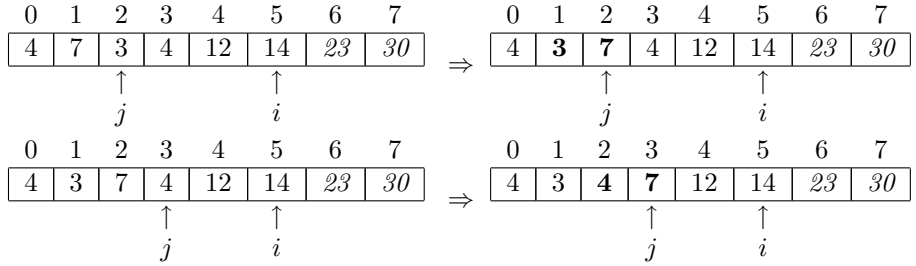
```
static void sortBule() {
  int t;
  for (int i = N1; i >= 0; i)
    for (int j = 1; j <= i; ++j)
      if (a[j1] > a[j]) {
        t = a[j1]; a[j1] = a[j]; a[j] = t;
      }
}
```



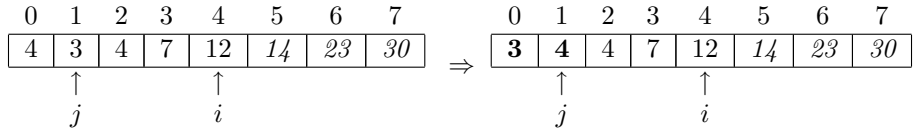
După prima parcurgere **30** a ajuns pe locul său (ultimul loc în vector).



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:

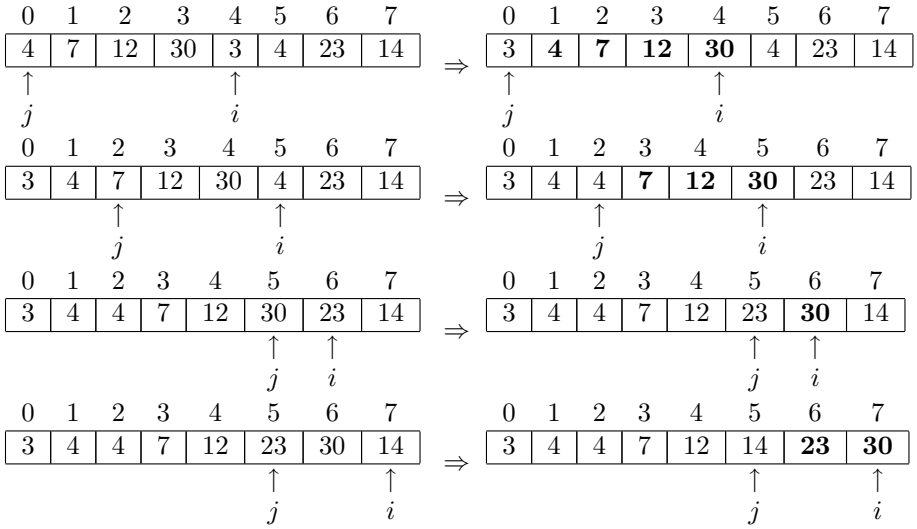
Inserarea lui **12** nu necesită deplasări de elemente.

0	1	2	3	4	5	6	7
7	12	4	30	3	4	23	14
↑		↑					
j		i					

$\Rightarrow$

0	1	2	3	4	5	6	7
4	**7**	**12**	30	3	4	23	14
↑		↑					
j		i					

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 π 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 pană 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;
    }
  }
}
```

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
}
```

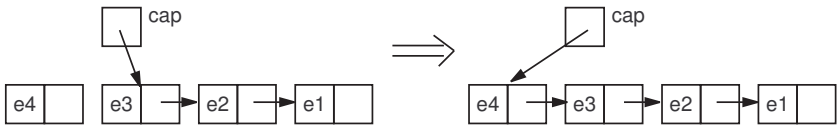


Figura 11.1: Adăugarea unui element în listă

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;
}
```

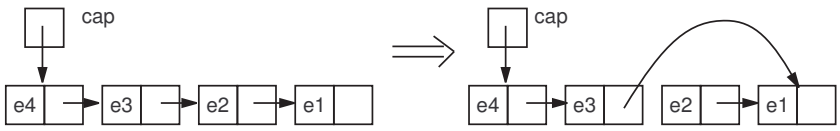


Figura 11.2: Eliminarea unui element din listă

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;
  }
}
```

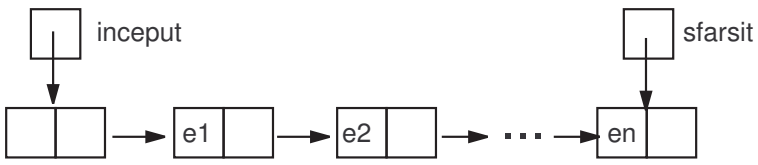


Figura 11.3: Coadă de aşteptare implementată ca listă

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.

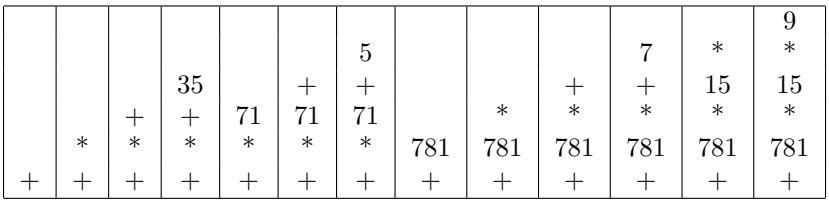
```
static int calcul (char a, int x, int y) {
  switch (a) {
    case '+': return x + y;
    case '*': return x * y;
  }
  return 1;
}
```

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.

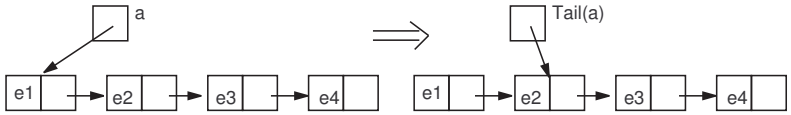


Figura 11.4: Suprimarea primului element din listă

```
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.

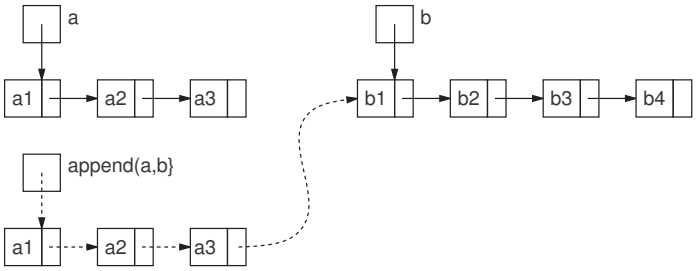


Figura 11.5: Concatenarea a două liste prin *append*

```
static Lista append (Lista a, Lista b) {
  if (a == null)
    return b;
  else
    return adauga(a.info, append (a.next, b)) ;
}
```

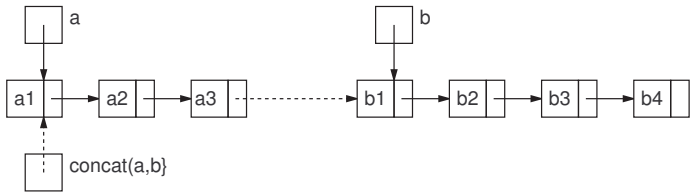


Figura 11.6: Concatenarea a două liste prin *concat*

```
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;
}
```

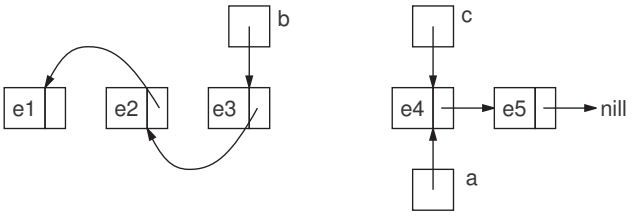


Figura 11.7: Transformarea listei prin inversarea legăturilor

```
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.

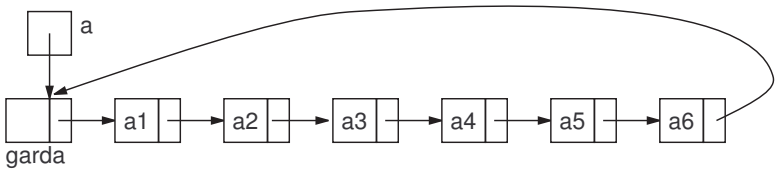


Figura 11.8: Listă circulară cu gardă

```
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 $P_1, ..., P_k$ de dimensiuni $n_1, ..., n_k$
      divideEtImpera(P_1, n_1, S_1)
      ...
      divideEtImpera(P_a, n_k, S_k)
      combină $S_1, ..., S_k$ 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 - Merge sort
- dreptunghi de arie maximă în placa cu găuri, etc

12.3.1 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).

```
class QuickSort
{
  static int x[]={3,5,2,6,4,1,8,2,4,3,5,3};

  public static void main(String[]args)
  {
```

```
    quickSort(0,x.length-1,x);
    for(int i=0;i<x.length;i++) System.out.print(x[i]+" ");
  }//main

  static void quickSort(int st,int dr,int a[])
  {
    int i=st, j=dr, x, temp;
    x=a[(st+dr)/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(st<j) quickSort(st,j,a);
    if(i<dr) quickSort(i,dr,a);
  }// quickSort(...)
}//class
```

Aceasta metoda are complexitatea $n \log n$, în practică (în medie)!

12.3.2 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 MergeSort
{
  static int x[]={3,5,2,6,4,1,8,2,4,3,5,3};

  public static void main(String[]args)
```

```
  {
    mergeSort(0,x.length-1,x);
    for(int i=0;i<x.length;i++) System.out.print(x[i]+" ");
  }//main

  public static int[] mergeSort(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;
      mergeSort(st,m,a);
      mergeSort(m+1,dr,a);
      interclasare(st,m,dr,a);
    }
    return a;
  }// mergeSort(...)

  public static int[] interclasare(int st,int m,int dr,int a[])
  {
    // interclasare pozitii st,...,m cu m+1,...,dr
    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(...)
}//class
```

12.3.3 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.

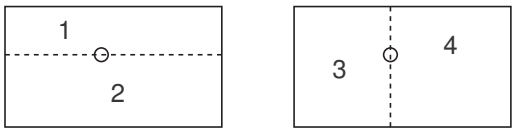


Figura 12.1: Dreptunghi de arie maximă în placa cu găuri

```
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(gasit)
    {
      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; }
  }
}
```

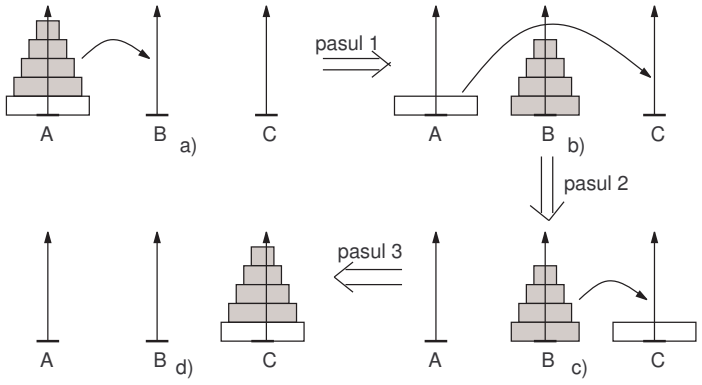
12.3.4 Turnurile din Hanoi

Se dau trei tije verticale **A**, **B2** şi **C**. Pe tija **A** se găsesc n discuri de diametre diferite, aranjate în ordine descrescătoare a diametrelor de la bază spre vârf. Se cere să se găsească o strategie de mutare a discurilor de pe tija **A** pe tija **C** respectând următoarele reguli:

• la un moment dat se va muta un singur disc (cel care se află deasupra celorlalte discuri pe o tijă);

• un disc poate fi aşezat doar peste un disc de diametru mai mare decât al să sau pe o tijă goală.

Împărţim problema astfel:
• se mută primele $n-1$ discuri de pe tija *sursă* **A** pe tija *intermediară* **B**
• se mută ultimul disc de pe tija *sursă* **A** pe tija *destinaţie* **C**
• se mută cele $n-1$ discuri de pe tija *intermediară* **B** pe tija *destinaţie* **C**




```
import java.io.*;
class Hanoi
{
  static int nrMutare=0;
  public static void main(String[] args) throws IOException
  {
    int nrDiscuri;

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    System.out.print("Introduceti numarul discurilor: ");
    nrDiscuri=Integer.parseInt(br.readLine());

    solutiaHanoi(nrDiscuri, 'A', 'B', 'C');
  }// main()

  static void solutiaHanoi(int nrDiscuri, char tijaSursa,
                           char tijaIntermediara, char tijaDestinatie)
  {
    if(nrDiscuri==1)
```

```
      System.out.println(++nrMutare + " Disc 1 " +
                         tijaSursa + " ==> "+ tijaDestinatie);
    else
    {
      solutiaHanoi(nrDiscuri-1,tijaSursa,tijaDestinatie,tijaIntermediara);

      System.out.println(++nrMutare + " Disk " + nrDiscuri +
                         " " + tijaSursa + " --> "+ tijaDestinatie);
      solutiaHanoi(nrDiscuri-1,tijaIntermediara,tijaSursa,tijaDestinatie);
    }
  }// solutiaHanoi()
}// class

Introduceti numarul discurilor: 4
 1 Disc 1 A ==> B
 2 Disk 2 A --> C
 3 Disc 1 B ==> C
 4 Disk 3 A --> B
 5 Disc 1 C ==> A
 6 Disk 2 C --> B
 7 Disc 1 A ==> B
 8 Disk 4 A --> C
 9 Disc 1 B ==> C
10 Disk 2 B --> A
11 Disc 1 C ==> A
12 Disk 3 B --> C
13 Disc 1 A ==> B
14 Disk 2 A --> C
15 Disc 1 B ==> C

import java.io.*;
class HanoiDisc
{
  static int nrMutare=0;
  static final int MAX_NR_DISCURI=9;
  static int[] a=new int[MAX_NR_DISCURI],
               b=new int[MAX_NR_DISCURI],
               c=new int[MAX_NR_DISCURI];
  static int na,nb,nc;         // na = nr. discuri pe tija A; etc. ...
  static int discMutat;

  public static void main(String[] args) throws IOException
  {
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

```
    int nrDiscuri=0;
    while((nrDiscuri<1)||(nrDiscuri>MAX_NR_DISCURI))
    {
      System.out.print("Numar discuri"+"[1<=...<="+MAX_NR_DISCURI+"]: ");
      nrDiscuri = Integer.parseInt(br.readLine());
    }
    na=nrDiscuri;
    nb=0;
    nc=0;
    for(int k=0;k<na;k++) a[k]=na-k;
    afisareDiscuri(); System.out.println();
    solutiaHanoi(nrDiscuri, 'A', 'B', 'C');
}// main()

  static void solutiaHanoi(int nrDiscuri, char tijaSursa,
                           char tijaIntermediara, char tijaDestinatie)
  {
    if (nrDiscuri==1)
    {
      mutaDiscul(tijaSursa, tijaDestinatie);
      System.out.print(++nrMutare + " Disc " + "1" + " " +
                       tijaSursa + " ==> "+ tijaDestinatie + " ");
      afisareDiscuri(); System.out.println();
    }
    else
    {
      solutiaHanoi(nrDiscuri-1,tijaSursa,tijaDestinatie,tijaIntermediara);
      mutaDiscul(tijaSursa, tijaDestinatie);
      System.out.print(++nrMutare + " Disc " + nrDiscuri +
                       " " + tijaSursa + " --> "+ tijaDestinatie + " ");
      afisareDiscuri();
      System.out.println();
      solutiaHanoi(nrDiscuri-1, tijaIntermediara, tijaSursa, tijaDestinatie);
    }
  } // solutiaHanoi()

  static void mutaDiscul(char tijaSursa, char tijaDestinatie)
  {
    switch (tijaSursa)
    {
      case 'A': discMutat=a[(na--)-1]; break;
      case 'B': discMutat=b[(nb--)-1]; break;
      case 'C': discMutat=c[(nc--)-1];
    }
```

```
    switch (tijaDestinatie)
    {
      case 'A': a[(++na)-1]=discMutat; break;
      case 'B': b[(++nb)-1]=discMutat; break;
      case 'C': c[(++nc)-1]=discMutat;
    }
  }// mutaDiscul()

  static void afisareDiscuri()
  {
    System.out.print("  A(");
    for(int i=0;i<na;i++) System.out.print(a[i]);
    System.out.print(") ");
    System.out.print("  B(");
    for(int i=0;i<nb;i++) System.out.print(b[i]);
    System.out.print(") ");
    System.out.print("  C(");
    for(int i=0;i<nc;i++) System.out.print(c[i]);
    System.out.print(") ");
  }// afisareDiscuri()
}// class
```

```
Numar discuri[1<=...<=9]: 4
                    A(4321) B()     C()
 1 Disc 1 A ==> B   A(432)  B(1)    C()
 2 Disc 2 A --> C   A(43)   B(1)    C(2)
 3 Disc 1 B ==> C   A(43)   B()     C(21)
 4 Disc 3 A --> B   A(4)    B(3)    C(21)
 5 Disc 1 C ==> A   A(41)   B(3)    C(2)
 6 Disc 2 C --> B   A(41)   B(32)   C()
 7 Disc 1 A ==> B   A(4)    B(321)  C()
 8 Disc 4 A --> C   A()     B(321)  C(4)
 9 Disc 1 B ==> C   A()     B(32)   C(41)
10 Disc 2 B --> A   A(2)    B(3)    C(41)
11 Disc 1 C ==> A   A(21)   B(3)    C(4)
12 Disc 3 B --> C   A(21)   B()     C(43)
13 Disc 1 A ==> B   A(2)    B(1)    C(43)
14 Disc 2 A --> C   A()     B(1)    C(432)
15 Disc 1 B ==> C   A()     B()     C(4321)
```

12.3.5 Înjumătăţire repetată

Se consideră un şir de numere naturale $x_1, x_2, ..., x_n$ şi o succesiune de decizii $d_1, d_2, ...$ unde $d_i \in \{S, D\}$ au următoarea semnificaţie: la pasul i, şirul rămas în acel moment se împarte în două subşiruri cu număr egal elemente (dacă numărul de elemente este impar, elementul situat la mijloc se elimină) şi se elimină jumătatea din partea stângă a şirului dacă $d_i = S$ (dacă $d_i = D$ se elimină jumătatea din partea dreptă).

Deciziile se aplică în ordine, una după alta, până când rămâne un singur element. Se cere acest element.

```
class Injumatatire1
{
  static int n=10;
  static char[] d={'X','S','D','S','S'}; // nu folosesc d_0 ... !

  public static void main(String[]args)
  {
    int st=1,dr=n,m,i=1;
    boolean impar;

    while(st<dr)
    {
      System.out.println(st+" ... "+dr+" elimin "+d[i]);
      m=(st+dr)/2;
      if((dr-st+1)%2==1) impar=true; else impar=false;

      if(d[i]=='S')      // elimin jumatatea stanga
      {
        st=m+1;
      }
      else               // elimin jumatatea dreapta
      {
        dr=m;
        if(impar) dr--;
      }
      i++;
      System.out.println(st+" ... "+dr+" a ramas! \n");
    }
  }//main
}//class

1 ... 10 elimin S
6 ... 10 a ramas!

6 ... 10 elimin D
```

```
6 ... 7 a ramas!

6 ... 7 elimin S
7 ... 7 a ramas!
```

Toate elementele care pot rămâne:

```
class Injumatatire2
{
  static int n=10;
  static char[] d=new char[10];

  public static void main(String[]args)
  {
    divide(1,n,1);
  }//main

  static void divide(int st0,int dr0, int i)
  {
    int st,dr,m;
    boolean impar;

    if(st0==dr0)
    {
      for(m=1;m<=i;m++) System.out.print(d[m]);
      System.out.println(" --> "+st0);
      return;
    }

    m=(st0+dr0)/2;
    if((dr0-st0+1)%2==1) impar=true; else impar=false;

    // elimin jumatatea stanga
    st=m+1;
    d[i]='S';
    if(st<=dr0) divide(st,dr0,i+1);

    // elimin jumatatea dreapta
    dr=m;
    if(impar) dr--;
    d[i]='D';
    if(st0<=dr) divide(st0,dr,i+1);
  }// divide(...)
}//class
```

```
SSS  --> 10
SSD  --> 9
SDS  --> 7
SDD  --> 6
DSS  --> 5
DSD  --> 4
DDS  --> 2
DDD  --> 1
```

Capitolul 13

Algoritmi BFS-Lee

13.1 Prezentare generală

```
import java.io.*;
class BFS     // nodurile sunt de la 1 la n
{             // distanta minima dintre nod "sursa" si nod "dest"
  static final int oo=0x7fffffff;          // infinit
  static final int WHITE=0, GRAY=1, BLACK=2;
  static int[][] a;      // matricea de adiacenta
  static int[] color;    // pentru bfs
  static int[] p;        // predecesor
  static int[] d;        // distante catre sursa
  static int[] q;        // coada
  static int ic;   // inceput coada = prima pozitie ocupata din care scot !
  static int sc;   // sfarsit coada = prima pozitie libera pe care voi pune !
  static int n,m;  // varfuri, muchii

  public static void main(String[] args) throws IOException
  {
    int i,j,k,nods,nodd;               // nod_sursa, nod_destinatie

    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("bfs.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("bfs.out")));
```

```
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); nods=(int)st.nval;
    st.nextToken(); nodd=(int)st.nval;

    a=new int[n+1][n+1];
    color=new int[n+1];
    p=new int[n+1];
    d=new int[n+1];
    q=new int[m+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=1;
      a[j][i]=1;
    }

    bfs(nods,nodd);

    System.out.println("Distanta("+nods+","+nodd+") = "+d[nodd]);
    System.out.print("drum : "); drum(nodd); System.out.println();
    out.close();
  }//main

  static void videzcoada()
  {
    ic=0;
    sc=0;  // coada : scot <-- icoada...scoada <-- introduc
  }

  static boolean coadaEsteVida()
  {
    return (sc==ic);
  }

  static void incoada(int v)
  {
    q[sc++]=v;
    color[v]=GRAY;
  }

  static int dincoada()
```

```
{
  int v=q[ic++];
  color[v]=BLACK;
  return v;
}

static void bfs(int start, int fin)
{
  int u, v;
  for(u=1; u<=n; u++)
  {
    color[u]=WHITE;
    d[u]=oo;
  }
  color[start]=GRAY;
  d[start]=0;

  videzcoada();
  incoada(start);
  while(!coadaEsteVida())
  {
    u=dincoada();

    // Cauta nodurile albe v adiacente cu nodul u si pun v in coada
    for(v=1; v<=n; v++)
    {
      if(a[u][v]==1)          // v in Ad[u]
      {
        if(color[v]==WHITE)   // neparcurs deja
        {
          color[v]=GRAY;
          d[v]=d[u]+1;
          p[v]=u;
          if(color[fin]!=WHITE) break; // optimizare; ies din for
          incoada(v);
        }
      }
    }//for

    color[u]=BLACK;
    if(color[fin]!=WHITE) break;  // am ajuns la nod_destinatie
  }//while
}//bfs
```

```
  static void drum(int u) // nod_sursa ---> nod_destinatie
  {
    if(p[u]!=0) drum(p[u]);
    System.out.print(u+" ");
  }
}//class

/*
9 12 2 8        Distanta(2,8) = 4
2 5             drum : 2 3 4 7 8
1 2
2 3
3 1
3 4
4 5
5 6
6 4
7 8
8 9
9 7
7 4
*/
```

13.2 Probleme rezolvate

13.2.1 Romeo şi Julieta - OJI2004 clasa a X-a

În ultima ecranizare a celebrei piese shakespeariene Romeo şi Julieta trăiesc într-un oraş modern, comunică prin e-mail şi chiar învaţă să programeze. Într-o secvenţă tulburătoare sunt prezentate frământările interioare ale celor doi eroi încercând fără succes să scrie un program care să determine un punct optim de întâlnire.

Ei au analizat harta oraşului şi au reprezentat-o sub forma unei matrice cu n linii şi m coloane, în matrice fiind marcate cu spaţiu zonele prin care se poate trece (străzi lipsite de pericole) şi cu X zonele prin care nu se poate trece. De asemenea, în matrice au marcat cu R locul în care se află locuinţa lui Romeo, iar cu J locul în care se află locuinţa Julietei.

Ei se pot deplasa numai prin zonele care sunt marcate cu spaţiu, din poziţia curentă în oricare dintre cele 8 poziţii învecinate (pe orizontală, verticală sau diagonale).

Cum lui Romeo nu îi place să aştepte şi nici să se lase aşteptat n-ar fi tocmai bine, ei au hotărât că trebuie să aleagă un punct de întâlnire în care atât Romeo, cât şi Julieta să poată ajunge în acelaşi timp, plecând de acasă. Fiindcă la întâlniri amândoi vin într-un suflet, ei estimează timpul necesar pentru a ajunge la întâlnire prin numărul de elemente din matrice care constituie drumul cel mai scurt de acasă până la punctul de întâlnire. Şi cum probabil există mai multe puncte de întâlnire posibile, ei vor să îl aleagă pe cel în care timpul necesar pentru a ajunge la punctul de întâlnire este minim.

Cerinţă

Scrieţi un program care să determine o poziţie pe hartă la care Romeo şi Julieta pot să ajungă în acelaşi timp. Dacă există mai multe soluţii, programul trebuie să determine o soluţie pentru care timpul este minim.

Datele de intrare

Fişierul de intrare **rj.in** conţine:

– pe prima linie numerele naturale NM, care reprezintă numărul de linii şi respectiv de coloane ale matricei, separate prin spaţiu;

– pe fiecare dintre următoarele N linii se află M caractere (care pot fi doar R, J, X sau spaţiu) reprezentând matricea.

Datele de ieşire

Fişierul de ieşire **rj.out** va conţine o singură linie pe care sunt scrise trei numere naturale separate prin câte un spaţiu $tmin\ x\ y$, având semnificaţia:

– $x\ y$ reprezintă punctul de întâlnire (x - numărul liniei, y - numărul coloanei);

– $tmin$ este timpul minim în care Romeo (respectiv Julieta) ajunge la punctul de întâlnire.

Restricţii şi precizări

- $1 < N, M < 101$
- Liniile şi coloanele matricei sunt numerotate începând cu 1.
- Pentru datele de test există întotdeauna soluţie.

Exemple

rj.in

```
5 8
XXR  XXX
 X  X  X
J X X  X
      XX
XXX XXXX
```

rj.out

```
4 4 4
```

Explicaţie:

Traseul lui Romeo poate fi: (1,3), (2,4), (3,4), (4,4). Timpul necesar lui Romeo pentru a ajunge de acasă la punctul de întâlnire este 4.

Traseul Julietei poate fi: (3,1), (4,2), (4,3), (4,5). Timpul necesar Julietei pentru a ajunge de acasă la punctul de întâlnire este deasemenea 4.

În plus 4, este punctul cel mai apropiat de ei cu această proprietate.

Timp maxim de executare: 1 secundă/test

Indicaţii de rezolvare *

Mihai Stroe, Ginfo nr. 14/4 aprilie 2004

Problema se rezolvă folosind *algoritmul lui Lee*.

Se aplică acest algoritm folosind ca puncte de start poziţia lui Romeo şi poziţia Julietei.

Vom folosi o matrice D în care vom pune valoarea 1 peste tot pe unde nu se poate trece, valoarea 2 în poziţia în care se află Romeo iniţial, valoarea 3 în poziţia în care se află Julieta iniţial şi valoarea 0 în rest.

La fiecare pas k vom parcurge această matrice şi în poziţiile vecine celor care au valoarea 2 vom pune valoarea 2, dacă acestea au valoarea 0 sau 3. Dacă o poziţie are valoare 3, înseamnă că la un moment de timp anterior Julieta se putea afla în poziţia respectivă. La acelaşi pas k vom mai parcurge matricea o dată şi în poziţiile vecine celor care au valoarea 3 vom pune valoarea 3, dacă acestea au valoarea 0.

Dacă la pasul k poziţia vecină uneia care are valoarea 3, are valoarea 2, atunci ne vom opri şi k reprezintă momentul minim de timp după care cei doi se întâlnesc, iar poziţia care are valoare 2 reprezintă locul întâlnirii.

La prima vedere s-ar părea că numărul k nu reprezintă momentul de timp minim la care cei doi se întâlnesc. Vom demonstra că algoritmul este corect prin metoda reducerii la absurd. Pentru aceasta avem în vedere că poziţiile marcate cu 2 reprezintă toate locurile în care se poate afla Romeo după cel mult k paşi, iar cele marcate cu 2 reprezintă toate locurile în care se poate afla Julieta după cel mult k paşi. Dacă k nu reprezintă momentul de timp minim la care cei doi se întâlnesc înseamnă că acesta a fost determinat mai devreme şi algoritmul s-a oprit deja.

Analiza complexităţii

Ordinul de complexitate al operaţiei de citire a datelor de intrare este $O(MN)$.

Ordinul de complexitate al acestui algoritm este $O(kMN)$, unde k reprezintă momentul în care cei doi se întâlnesc.

Ordinul de complexitate al operaţiei de scriere a rezultatului este $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(kMN)$.

Rezolvare detaliată

Codul sursă *

```
import java.io.*;
class RJ
{
  static final int zid=10000;
  static int m,n;
  static int[][] xr,xj;
  static int[] qi=new int[5000];  // coada sau coada circulara mai bine !
  static int[] qj=new int[5000];  // coada

  static int ic, sc;              // ic=inceput coada

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j,ir=0,jr=0,ij=0,jj=0, imin, jmin, tmin;
    String s;

    BufferedReader br=new BufferedReader(new FileReader("rj.in"));
    StreamTokenizer st=new StreamTokenizer(br);
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("rj.out")));

    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;

    xr=new int[m+2][n+2];         // matrice bordata cu zid !
    xj=new int[m+2][n+2];         // matrice bordata cu zid !

    br.readLine();                // citeste CRLF !!!
    for(i=1;i<=m;i++)
    {
      s=br.readLine();
      for(j=1;j<=n;j++)
        if(s.charAt(j-1)=='X') xr[i][j]=xj[i][j]=zid;                // zid!
        else if(s.charAt(j-1)=='R') {ir=i; jr=j; xr[i][j]=1;}
```

```
        else if(s.charAt(j-1)=='J') {ij=i; jj=j; xj[i][j]=1;}
    }

    for(i=0;i<=m+1;i++) xr[i][0]=xr[i][n+1]=xj[i][0]=xj[i][n+1]=zid; // E si V
    for(j=0;j<=n+1;j++) xr[0][j]=xr[m+1][j]=xj[0][j]=xj[m+1][j]=zid; // N si S

    ic=sc=0;                        // coada vida;
    qi[sc]=ir; qj[sc]=jr; sc++;   // (ir,jr) --> coada
    while(ic!=sc)
    {
      i=qi[ic]; j=qj[ic]; ic++;  // scot din coada
      fill(xr,i,j);
    }

    ic=sc=0;                       // coada vida;
    qi[sc]=ij; qj[sc]=jj; sc++; // (ij,jj) --> coada
    while(ic!=sc)
    {
      i=qi[ic]; j=qj[ic]; ic++;  // scot din coada
      fill(xj,i,j);
    }

    tmin=10000;
    imin=jmin=0;
    for(i=1;i<=m;i++)
      for(j=1;j<=n;j++)
        if(xr[i][j]==xj[i][j])
        if(xj[i][j]!=0)              // pot exista pozitii ramase izolate !
        if(xr[i][j]<tmin) {tmin=xr[i][j]; imin=i; jmin=j;}

    out.println(tmin+" "+imin+" "+jmin);
    out.close();
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()

  static void fill(int[][] x,int i, int j)
  {
    int t=x[i][j];      // timp
    if(x[i-1][j]==0)    {x[i-1][j]=t+1;   qi[sc]=i-1; qj[sc]=j;   sc++;} // N
    if(x[i-1][j+1]==0)  {x[i-1][j+1]=t+1; qi[sc]=i-1; qj[sc]=j+1; sc++;} // NE
    if(x[i-1][j-1]==0)  {x[i-1][j-1]=t+1; qi[sc]=i-1; qj[sc]=j-1; sc++;} // NV

    if(x[i+1][j]==0)    {x[i+1][j]=t+1;   qi[sc]=i+1; qj[sc]=j;   sc++;} // S
```

```
    if(x[i+1][j+1]==0)  {x[i+1][j+1]=t+1; qi[sc]=i+1; qj[sc]=j+1; sc++;} // SE
    if(x[i+1][j-1]==0)  {x[i+1][j-1]=t+1; qi[sc]=i+1; qj[sc]=j-1; sc++;} // SV

    if(x[i][j+1]==0)    {x[i][j+1]=t+1;   qi[sc]=i;   qj[sc]=j+1; sc++;} // E
    if(x[i][j-1]==0)    {x[i][j-1]=t+1;   qi[sc]=i;   qj[sc]=j-1; sc++;} // V
  }// fil(...)
}// class
```

13.2.2 Sudest - OJI2006 clasa a X-a

Fermierul Ion deţine un teren de formă pătrată, împărţit în $N \times N$ pătrate de latură unitate, pe care cultivă cartofi. Pentru recoltarea cartofilor fermierul foloseşte un robot special proiectat în acest scop. Robotul porneşte din pătratul din stânga sus, de coordonate $(1, 1)$ şi trebuie să ajungă în pătratul din dreapta jos, de coordonate (N, N).

Traseul robotului este programat prin memorarea unor comenzi pe o cartelă magnetică. Fiecare comandă specifică direcţia de deplasare (sud sau est) şi numărul de pătrate pe care le parcurge în direcţia respectivă. Robotul strânge recolta doar din pătratele în care se opreşte între două comenzi.

Din păcate, cartela pe care se află programul s-a deteriorat şi unitatea de citire a robotului nu mai poate distinge direcţia de deplasare, ci numai numărul de paşi pe care trebuie să-i facă robotul la fiecare comandă. Fermierul Ion trebuie să introducă manual, pentru fiecare comandă, direcţia de deplasare.

Cerinţă

Scrieţi un program care să determine cantitatea maximă de cartofi pe care o poate culege robotul, în ipoteza în care Ion specifică manual, pentru fiecare comandă, direcţia urmată de robot. Se va determina şi traseul pe care se obţine recolta maximă.

Datele de intrare

Fişierul de intrare **sudest.in** are următoarea structură:

- Pe linia 1 se află numărul natural N, reprezentând dimensiunea parcelei de teren.
- Pe următoarele N linii se află câte N numere naturale, separate prin spaţii, reprezentând cantitatea de cartofi din fiecare pătrat unitate.
- Pe linia $N + 2$ se află un număr natural K reprezentând numărul de comenzi aflate pe cartela magnetică.
- Pe linia $N + 3$ se află K numerele naturale C_1, ...,C_K, separate prin spaţii, reprezentând numărul de paşi pe care trebuie să-i efectueze robotul la fiecare comandă.

Datele de ieşire

Fişierul de iesire **sudest.out** va conţine pe prima linie cantitatea maximă de cartofi recoltată de robot. Pe următoarele $K+1$ linii vor fi scrise, în ordine, coordonatele pătratelor unitate ce constituie traseul pentru care se obţine cantitate maximă de cartofi, câte un pătrat unitate pe o linie. Coordonatele scrise pe aceeaşi linie vor fi separate printr-un spaţiu. Primul pătrat de pe traseu va avea coordonatele 11, iar ultimul va avea coordonatele NN. Dacă sunt mai multe trasee pe care se obţine o cantitate maximă de cartofi recoltată se va afişa unul dintre acestea.

Restricţii şi precizări

- $5 \leq N \leq 100$
- $2 \leq K \leq 2 * N - 2$
- $1 \leq C_1, ..., C_K \leq 10$
- Cantitatea de cartofi dintr-un pătrat de teren este număr natural între 0 şi 100.
- Pentru fiecare set de date de intrare se garantează că există cel puţin un traseu.
- Se consideră că robotul strânge recolta şi din pătratul de plecare $(1, 1)$ şi din cel de sosire (N, N).
- Pentru determinarea corectă a cantităţii maxime recoltate se acordă 50% din punctajul alocat testului respectiv; pentru cantitate maximă recoltată şi traseu corect se acordă 100%.

Exemplu

sudest.in	**sudest.out**	**Explicaţii**
6	29	Un alt traseu posibil este:
1 2 1 0 4 1	1 1	1 1
1 3 3 5 1 1	3 1	1 3
2 2 1 2 1 10	5 1	1 5
4 5 3 9 2 6	6 1	2 5
1 1 3 2 0 1	6 5	6 5
10 2 4 6 5 10	6 6	6 6
5		dar costul său este $1+1+4+1+5+10=22$
2 2 1 4 1		

Timp maxim de execuţie/test: 1 secundă

Indicaţii de rezolvare *

soluţia comisiei

Reprezentarea informaţiilor

- N - numărul de linii
- K - numărul de comenzi
- $A[Nmax][Nmax]$; - memorează cantitatea de produs
- $C[Nmax][Nmax]$; - $C[i][j]$ = cantitatea maximă de cartofi culeasă pe un traseu ce porneşte din $(1,1)$ şi se termină în (i,j), respectând condiţiile problemei
- $P[Nmax][Nmax]$; - $P[i][j]$ = pasul la care am ajuns în poziţia i,j culegând o cantitate maximă de cartofi
- $Move[2*Nmax]$; - memorează cele K comenzi

Parcurg şirul celor k mutări. La fiecare mutare marchez poziţiile în care pot ajunge la mutarea respectivă.

Mai exact, parcurg toate poziţiile în care am putut ajunge la pasul precedent (cele marcate în matricea P corespunzător cu numărul pasului precedent) şi pentru fiecare poziţie verific dacă la pasul curent pot să execut mutarea la sud.

În caz afirmativ, verific dacă în acest caz obţin o cantitate de cartofi mai mare decât cea obţinută până la momentul curent (dacă da, reţin noua cantitate, şi marchez în matricea P poziţia în care am ajuns cu indicele mutării curente).

În mod similar procedez pentru o mutare spre est.

Codul sursă *

Variantă după soluţia oficială:

```
import java.io.*;
class Sudest1
{
  static StreamTokenizer st;
  static PrintWriter out;
  static final int Nmax=101;
  static int N, K;
  static int[][] A=new int[Nmax][Nmax]; // A[i][j]=cantitatea de cartofi
  static int[][] C=new int[Nmax][Nmax]; // C[i][j]=cantitatea maxima ...
  static int[][] P=new int[Nmax][Nmax]; // pas
  static int[]  Move=new int[2*Nmax];   // comenzile
```

```
public static void main(String[] args) throws IOException
{
  st=new StreamTokenizer( new BufferedReader(new FileReader("sudest.in")));
  out=new PrintWriter(new BufferedWriter(new FileWriter("sudest.out")));
  read_data();
  init();
  solve();
}// main(...)

static void read_data() throws IOException
{
  int i,j,cmd;
  st.nextToken(); N=(int)st.nval;
  for(i=1;i<=N;i++)
    for(j=1;j<=N;j++) { st.nextToken(); A[i][j]=(int)st.nval;}
  st.nextToken(); K=(int)st.nval;
  for(cmd=1;cmd<=K;cmd++) { st.nextToken(); Move[cmd]=(int)st.nval;}
}// read_data()

static void init()
{
  int i,j;
  for(i=1;i<=N;i++) for(j=1;j<=N;j++) {C[i][j]=-1; P[i][j]=255;}
}// init()

static boolean posibil(int x,int y) {return 1<=x && 1<=y && x<=N && y<=N;}

static void solve()
{
  int i,j, cmd;
  P[1][1]=0;
  C[1][1]=A[1][1];
  for(cmd=1; cmd<=K; cmd++)
    for(i=1; i<=N; i++)
      for(j=1; j<=N; j++)
        if(P[i][j]==cmd-1)
        {
          if(posibil(i+Move[cmd],j)) // SUD
          if(C[i][j]+A[i+Move[cmd]][j]>C[i+Move[cmd]][j])
          {
            P[i+Move[cmd]][j]=cmd;
            C[i+Move[cmd]][j]=C[i][j]+A[i+Move[cmd]][j];
          }
```

```
            if(posibil(i,j+Move[cmd])) // EST
              if(C[i][j]+A[i][j+Move[cmd]]>C[i][j+Move[cmd]])
              {
                P[i][j+Move[cmd]]=cmd;
                C[i][j+Move[cmd]]=C[i][j]+A[i][j+Move[cmd]];
              }
          }// if
    out.println(C[N][N]); drum(N,N,K); out.close();
  }// solve()

  static void drum(int x, int y, int pas)
  {
    int i;
    boolean gasit;
    if(x==1 && y==1) out.println("1 1");
    else
    {
      gasit=false;
      if(posibil(x,y-Move[pas]))
        if(C[x][y-Move[pas]]==C[x][y]-A[x][y] && P[x][y-Move[pas]]==pas-1)
        {
          drum(x,y-Move[pas],pas-1);
          out.println(x+" "+y);
          gasit=true;
        }
      if(!gasit)
        if(posibil(x-Move[pas],y))
          if(C[x-Move[pas]][y]==C[x][y]-A[x][y] && P[x-Move[pas]][y]==pas-1)
          {
            drum(x-Move[pas],y,pas-1);
            out.println(x+" "+y);
          }
    }// else
  }// drum(...)
}// class
```

Variantă folosind coadă:

```
import java.io.*; // 11, ...,nn --> 1,...,n*n pozitii in matrice !
class Sudest2
{
  static StreamTokenizer st;
  static PrintWriter out;
  static final int nmax=101;
  static int n, k;
```

```
static int[][] a=new int[nmax][nmax]; // a[i][j]=cantitatea de cartofi
static int[][] c=new int[nmax][nmax]; // c[i][j]=cantitatea maxima ...
static int[][] p=new int[nmax][nmax]; // pozitia anterioara optima
static int[]  move=new int[2*nmax];   // comenzile
static int ic,sc,scv,qmax=100;
static int[] q=new int[qmax]; // coada

public static void main(String[] args) throws IOException
{
  st=new StreamTokenizer(new BufferedReader(new FileReader("sudest.in")));
  out=new PrintWriter(new BufferedWriter(new FileWriter("sudest.out")));
  read_data();
  init();
  solve();
}// main(...)

static void read_data() throws IOException
{
  int i,j,cmd;
  st.nextToken(); n=(int)st.nval;
  for(i=1;i<=n;i++)
    for(j=1;j<=n;j++) { st.nextToken(); a[i][j]=(int)st.nval;}
  st.nextToken(); k=(int)st.nval;
  for(cmd=1;cmd<=k;cmd++) { st.nextToken(); move[cmd]=(int)st.nval;}
}// read_data()

static void init()
{
  int i,j;
  for(i=1;i<=n;i++) for(j=1;j<=n;j++) {c[i][j]=-1; p[i][j]=255;}
}// init()

static void solve()
{
  int i,j,ii,jj,cmd;
  p[1][1]=0; c[1][1]=a[1][1];
  ic=0; sc=1; q[ic]=1;  // pozitia [1][1] --> q
  scv=1;                // ultimul din coada la distanta 1 de [1][1]
  for(cmd=1; cmd<=k; cmd++)
  {
    while(ic!=scv)
    {
      i=(q[ic]-1)/n+1; j=(q[ic]-1)%n+1; ic=(ic+1)%qmax; // scot din coada
      // propag catre Sud
```

```
        ii=i+move[cmd];
        jj=j;
        if((ii>=1)&&(ii<=n))
          if(c[i][j]+a[ii][jj]>c[ii][jj])
          {
            c[ii][jj]=c[i][j]+a[ii][jj];
            p[ii][jj]=(i-1)*n+j;
            q[sc]=(ii-1)*n+jj; sc=(sc+1)%qmax; // pun in coada
          }

        // propag catre Est
        jj=j+move[cmd];
        ii=i;
        if((jj>=1)&&(jj<=n))
          if(c[i][j]+a[ii][jj]>c[ii][jj])
          {
            c[ii][jj]=c[i][j]+a[ii][jj];
            p[ii][jj]=(i-1)*n+j;
            q[sc]=(ii-1)*n+jj; sc=(sc+1)%qmax; // pun in coada
        }
      }// while
      scv=sc;
    }// for

    out.println(c[n][n]); drum(n,n); out.close();
  }// solve()

  static void drum(int i, int j)
  {
    if(i*j==0) return;
    drum((p[i][j]-1)/n+1,(p[i][j]-1)%n+1);
    out.println(i+" "+j);
  }// drum(...)
}// class
```

13.2.3 Muzeu - ONI2003 clasa a X-a

Sunteţi un participant la Olimpiada Naţională de Informatică. În programul olimpiadei intră şi câteva activităţii de divertisment. Una dintre ele este vizitarea unui muzeu. Acesta are o structură de matrice dreptunghiulară cu M linii şi N coloane; din orice cameră se poate ajunge în camerele vecine pe direcţiile nord, est, sud şi vest (dacă aceste camere există). Pentru poziţia (i, j) deplasarea spre nord presupune trecerea în poziţia $(i-1, j)$, spre est în $(i, j+1)$, spre sud în

$(i+1, j)$ şi spre vest în $(i, j-1)$.

Acest muzeu are câteva reguli speciale. Fiecare cameră este marcată cu un număr între 0 şi 10 inclusiv. Mai multe camere pot fi marcate cu acelaşi număr. Camerele marcate cu numărul 0 pot fi vizitate gratuit. Într-o cameră marcată cu numărul i ($i > 0$) se poate intra gratuit, dar nu se poate ieşi din ea decât dacă arătaţi supraveghetorului un bilet cu numărul i. Din fericire, orice cameră cu numărul i ($i > 0$) oferă spre vânzare un bilet cu numărul i; o dată cumpărat acest bilet, el este valabil în toate camerele marcate cu numărul respectiv. Biletele pot avea preţuri diferite, dar un bilet cu numărul i va avea acelaşi preţ în toate camerele în care este oferit spre vânzare.

Dumneavoastră intraţi în muzeu prin colţul de Nord-Vest (poziţia $(1, 1)$ a matricei) şi doriţi să ajungeţi la ieşirea situată în colţul de Sud-Est (poziţia (M, N) a matricei). O dată ajuns acolo primiţi un bilet gratuit care vă permite să vizitaţi tot muzeul.

Poziţiile $(1, 1)$ şi (M, N) sunt marcate cu numărul 0.

Cerinţă

Cunoscându-se structura muzeului, determinaţi o strategie de parcurgere a camerelor, astfel încât să ajungeţi în camera (M, N) plătind cât mai puţin. Dacă există mai multe variante, alegeţi una în care este parcurs un număr minim de camere (pentru a câ?tiga timp şi pentru a avea mai mult timp pentru vizitarea integrală a muzeului).

Date de intrare

Prima linie a fişierului de intrare **muzeu.in** conţine două numere întregi M şi N, separate printr-un spaţiu, numărul de linii, respectiv de coloane, al matricei care reprezintă muzeul. Următoarele M linii conţin structura muzeului; fiecare conţine N numere întregi între 0 şi 10 inclusiv, separate prin spaţii. Linia $M + 2$ conţine 10 numere întregi între 0 şi 10000 inclusiv, reprezentând costurile biletelor $1, 2, 3, ...10$ în această ordine.

Date de ieşire

În fişierul **muzeu.out** veţi afişa:

pe prima linie suma minimă necesară pentru a ajunge din $(1, 1)$ în (M, N);

pe a doua linie numărul minim de mutări L efectuate dintr-o cameră într-o cameră vecină, pentru a ajunge din $(1, 1)$ în (M, N);

pe a treia linie L caractere din multimea N, E, S, V reprezentând deplasări spre Nord, Est, Sud sau Vest.

Restricţii şi precizări

$2 \leq N \leq 50$

Exemplu:

muzeu.in	**muzeu.out**
5 6 0 0 0 0 0 2 0 1 1 1 4 3 0 1 0 0 0 0 0 1 5 1 0 0 0 0 0 1 0 0 1000 5 7 100 12 1000 1000 1000 1000 1000	12 9 EEEEESSSS

Timp maxim de executare: 1 secundă/test.

Indicaţii de rezolvare *

Mihai Stroe, GInfo nr. 13/6 - octombrie 2003

Se observă că numărul variantelor de cumpărare a biletelor (cumpăr / nu cumpăr biletul 1, biletul 2, ..., biletul 10) este $2^{10} = 1024$.

Se generează fiecare din aceste variante. Pentru o astfel de variantă, se calculează costul şi se transformă matricea iniţială într-o matrice cu 0 şi 1, în care 0 reprezintă o cameră pentru care se plăteşte biletul (sau nu e nevoie de bilet) şi 1 reprezintă o cameră pentru care nu se cumpăra bilet (deci în care nu se intră).

Pentru o astfel de matrice, problema determinării celui mai scurt drum de la $(1, 1)$ la (M, N) se rezolvă cu *algoritmul lui Lee*.

Se rezolvă această problemă pentru toate cele 1024 variante. Eventual, dacă la un moment dat există o soluţie cu un cost mai bun decât al variantei curente, aceasta nu mai este abordată.

Evident, problema determinării celui mai scurt drum se poate rezolva şi pe matricea iniţială, ţinând cont la fiecare pas de biletele cumpărate; cum *algoritmul lui Lee* este folosit de obicei pentru o matrice cu 0 şi 1, am folosit iniţial convenţia respectivă pentru a uşura înelegerea.

Se alege soluţia de cost minim şi, în caz de egalitate, cea cu număr minim de camere.

Analiza complexităţii

Operaţia de citire a datelor are ordinul de complexitate $O(M \cdot N)$.

Fie C numărul de numere de marcaj diferite din matrice. Pentru fiecare dintre cele 2^C variante, găsirea drumului minim cu *algoritmul lui Lee* are complexitatea $O(M \cdot N)$.

Operaţia de scriere a soluţiei are ordinul de complexitate $O(M \cdot N)$ pentru cazul cel mai defavorabil.

Ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(M \cdot N \cdot 2^C)$.

Algoritmul funcţionează datorită restricţiei impuse pentru C (cel mult 10 numere de marcaj). Considerând 2^C o constantă, ordinul de complexitate devine $O(M \cdot N)$.

Codul sursă

```
import java.io.*;
class Muzeu
{
  static final int qdim=200;        // dimensiune coada circulara
  static final int sus=1, dreapta=2, jos=3, stanga=4;

  static int m,n;                   // dimensiuni muzeu
  static int ncmin;                 // nr camere minim
  static int cc,cmin=50*50*10000/2; // cost curent/minim
  static boolean amAjuns;

  static int[][] x=new int[51][51];   // muzeu
  static int[][] c=new int[51][51];   // costul (1,1) --> (i,j)
  static int[][] d=new int[51][51];   // directii pentru "intoarcere"
  static int[][] dsol=new int[51][51];

  static int[] cb=new int[11];        // costuri bilete
  static boolean[] amBilet=new boolean[11];

  static int[] qi=new int[qdim];  // coada pentru i din pozitia (i,j)
  static int[] qj=new int[qdim];  // coada pentru j din pozitia (i,j)
  static int ic, sc;              // ic=inceput coada

  public static void main(String[] args) throws IOException
  {
    int i,j;
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("10-muzeu.in")));

    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;

    for(i=1;i<=m;i++)
    for(j=1;j<=n;j++) { st.nextToken(); x[i][j]=(int)st.nval; }

    for(i=1;i<=10;i++) {st.nextToken(); cb[i]=(int)st.nval; }

    amBilet[0]=true;    // 0 ==> gratuit
    for(i=0;i<=1023;i++)
    {
      bilete(i);
```

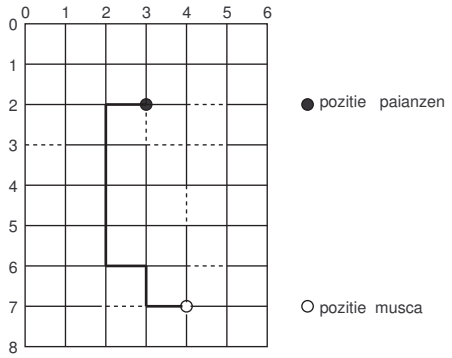
```
      cc=0; for(j=1;j<=10;j++) if(amBilet[j]) cc+=cb[j];
      if(cc>cmin) continue;

      amAjuns=false;
      matriceCosturi();

      if(!amAjuns) continue;
      if(cc>cmin) continue;
      if(cc<cmin) { cmin=cc; ncmin=c[m][n]; copieDirectii(); }
      else    // costuri egale
        if(c[m][n]<ncmin) { ncmin=c[m][n]; copieDirectii(); }
    }

    d=dsol; // schimbare "adresa" ... ("pointer"!) ...
    afisSolutia();
  }// main()

  static void bilete(int i)
  {
    int j;
    for(j=1;j<=10;j++)
    {
      if(i%2==1) amBilet[j]=true; else amBilet[j]=false;
      i/=2;
    }
  }// bilete(...)

  static void matriceCosturi()
  {
    int i,j;
    for(i=1;i<=m;i++)
    for(j=1;j<=n;j++) c[i][j]=0;         // curat "numai" traseul !
    ic=sc=0; // coada vida
    qi[sc]=1; qj[sc]=1; sc=(sc+1)%qdim;  // (1,1) --> coada circulara !
    c[1][1]=1;    // cost 1 pentru pozitia (1,1) (pentru marcaj!)
    while(ic!=sc) // coada nevida
    {
      i=qi[ic]; j=qj[ic]; ic=(ic+1)%qdim;
      vecini(i,j);
      if(amAjuns) break;
    }
  }//matriceCosturi()

  static void copieDirectii()
```

```
{
  int i,j;
  for(i=1;i<=m;i++) for(j=1;j<=n;j++) dsol[i][j]=d[i][j];
}// copieDirectii()

static void vecini(int i, int j)
{
  int t=c[i][j];    // "timp" = nr camere parcurse

  if((i-1>=1)&&(c[i-1][j]==0)&&amBilet[x[i-1][j]])  // N
  {
    c[i-1][j]=t+1; qi[sc]=i-1; qj[sc]=j; sc=(sc+1)%qdim;
    d[i-1][j]=jos;
    if((i-1==m)&&(j==n)) {amAjuns=true; return;}
  }

  if((j+1<=n)&&(c[i][j+1]==0)&&amBilet[x[i][j+1]]) // E
  {
    c[i][j+1]=t+1; qi[sc]=i; qj[sc]=j+1; sc=(sc+1)%qdim;
    d[i][j+1]=stanga;
    if((i==m)&&(j+1==n)) {amAjuns=true; return;}
  }

  if((i+1<=m)&&(c[i+1][j]==0)&&amBilet[x[i+1][j]])// S
  {
    c[i+1][j]=t+1; qi[sc]=i+1; qj[sc]=j; sc=(sc+1)%qdim;
    d[i+1][j]=sus;
    if((i+1==m)&&(j==n)) {amAjuns=true; return;}
  }

  if((j-1>=1)&&(c[i][j-1]==0)&&amBilet[x[i][j-1]]) // V
  {
    c[i][j-1]=t+1; qi[sc]=i; qj[sc]=j-1; sc=(sc+1)%qdim;
    d[i][j-1]=dreapta;
    if((i==m)&&(j-1==n)) {amAjuns=true; return;}
  }
}// vecini(...)

static void afisSolutia() throws IOException
{
  int i,j;
  PrintWriter out = new PrintWriter(
    new BufferedWriter(new FileWriter("muzeu.out")));
```

```
    out.println(cmin);
    out.println(ncmin-1);

    i=m; j=n;                  // (m,n) --> (1,1)
    while((i!=1)||(j!=1))      // folosesc "c" care nu mai e necesara !
    if(d[i][j]==sus) { c[i-1][j]=jos; i--; }
    else if(d[i][j]==jos) { c[i+1][j]=sus; i++;}
    else if(d[i][j]==dreapta) { c[i][j+1]=stanga; j++; }
    else if(d[i][j]==stanga) { c[i][j-1]=dreapta; j--; }
    else System.out.println("Eroare la traseu ... (m,n)-->(1,1)!");

    i=1; j=1;                    // (1,1) --> (m,n)
    while((i!=m)||(j!=n))
    {
      if(c[i][j]==sus) {out.print("N"); i--;}
      else if(c[i][j]==jos) {out.print("S"); i++;}
      else if(c[i][j]==dreapta) {out.print("E"); j++;}
      else if(c[i][j]==stanga) {out.print("V"); j--;}
      else System.out.println("Eroare la traseu ... (1,1)--(m,n)!");
    }
    out.close();
  }//afisSolutia()
}// class
```

13.2.4 Păianjen ONI2005 clasa a X-a

Un păianjen a ţesut o plasă, în care nodurile sunt dispuse sub forma unui caroiaj cu m linii (numerotate de la 0 la $m-1$) şi n coloane (numerotate de la 0 la $n-1$) ca în figură. Iniţial, oricare două noduri vecine (pe orizontală sau verticală) erau unite prin segmente de plasă de lungime 1. În timp unele porţiuni ale plasei s-au deteriorat, devenind nesigure. Pe plasă, la un moment dat, se găsesc păianjenul şi o muscă, în noduri de coordonate cunoscute.



Cerinţă

Să se determine lungimea celui mai scurt traseu pe care trebuie să-l parcurgă păianjenul, folosind doar porţiunile sigure ale plasei, pentru a ajunge la muscă. De asemenea, se cere un astfel de traseu.

Datele de intrare

Fişierul de intrare **paianjen.in** conţine:

– pe prima linie două numere naturale m n, separate printr-un spaţiu, reprezentând numărul de linii şi respectiv numărul de coloane ale plasei;

– pe a doua linie două numere naturale lp cp, separate printr-un spaţu, reprezentnd linia şi respectiv coloana nodului în care se află iniţial păianjenul;

– pe linia a treia două numere naturale lm cm separate printr-un spaţiu, reprezentând linia şi respectiv coloana pe care se află iniţial musca;

– pe linia a patra, un număr natural k, reprezentând numărul de porţiuni de plasă deteriorate;

– pe fiecare dintre următoarele k linii, câte patru valori naturale $l1$ $c1$ $l2$ $c2$, separate prin câte un spaţiu, reprezentând coordonatele capetelor celor k porţiuni de plasă deteriorate (linia şi apoi coloana pentru fiecare capăt).

Datele de ieşire

Fişierul de ieşire **paianjen.out** va conţine pe prima linie un număr natural min reprezentând lungimea drumului minim parcurs de păianjen, exprimat în număr de segmente de lungime 1. Pe următoarele $min+1$ linii sunt scrise nodurile prin care trece păianjenul, câte un nod pe o linie. Pentru fiecare nod sunt scrise linia şi coloana pe care se află, separate printr-un spaţiu.

Restricţii şi precizări

- $1 \leq m, n \leq 140$
- $1 \leq k \leq 2 * (m * n - m - n + 1)$
- Lungimea drumului minim este cel mult 15000
- Pentru datele de test există întotdeauna soluţie. Dacă problema are mai multe soluţii, se va afişa una singură.

• Porţiunile nesigure sunt specificate în fişierul de intrare într-o ordine oarecare. Oricare două porţiuni nesigure orizontale se pot intersecta cel mult într-un capăt. De asemenea, oricare două porţiuni nesigure verticale se pot intersecta cel mult într-un capăt.

• Se acordă 30% din punctaj pentru determinarea lungimii drumului minim şi 100% pentru rezolvarea ambelor cerinţe.

Exemplu

paianjen.in	**paianjen.out**	*Explicaţie*
9 7 2 3 7 4 8 2 4 2 5 2 3 3 3 3 0 3 1 3 3 3 5 4 4 5 4 6 4 6 5 6 5 7 5 7 2 7 3	8 2 3 2 2 3 2 4 2 5 2 6 2 6 3 7 3 7 4	Problema corespunde figurii de mai sus. Traseul optim este desenat cu linie groasă, iar porţiunile nesigure sunt desenate punctat.

Timp maxim de execuţie/test: 1 secundă pentru Windows şi 0.1 secunde pentru Linux.

Indicaţii de rezolvare *

prof. Carmen Popescu, C. N. "Gh. Lazăr" Sibiu

Plasa de păianjen se memorează într-o matrice A cu M linii şi N coloane, fiecare element reprezentând un nod al plasei. $A[i, j]$ va codifica pe patru biţi direcţiile în care se poate face deplasarea din punctul (i, j): bitul 0 este 1 dacă păianjenul se poate deplasa în sus, bitul 1 este 1 dacă se poate deplasa la dreapta, bitul 2 - în jos, bitul 3 - la stânga.

Rezolvarea se bazează pe parcurgerea matriciei şi reţinerea nodurilor parcurse într-o *structură de date de tip coadă* (parcurgere BF - algoritm Lee).

Drumul minim al păianjenului se reţine într-o altă matrice B, unde $B[i, j]$ este 0 dacă nodul (i, j) nu este atins, respectiv o valoare pozitivă reprezentând pasul la care a ajuns paianjenul în drumul lui spre muscă. Deci elementul $B[lm, cm]$ va conţine lungimea drumului minim.

Reconstituirea drumului minim se face pornind de la poziţia muştei, utilizând, de asemenea un algoritm de tip BF, cu oprirea căutării în jurul nodului curent în momentul detectării unui nod de pas mai mic cu o unitate decât cel al nodului curent.

TESTE

#	m	n	k	min	Obs
0	10	8	11	16	dimensiune mică, fire puţine rupte
1	10	8	46	74	soluţie unică, foarte multe fire rupte
2	2	2	0	2	caz particular, nici un fir rupt
3	140	140	20	278	fire puţine rupte, dimensiune mare
4	131	131	2000	103	multe fire rupte, dimensiune mare
5	100	130	12771	12999	traseu în spirală solutie unică
6	100	7	23	15	traseu scurt, greu de găsit cu backtracking
7	138	138	9381	9050	multe fire rupte, drum în "serpentine"
8	140	140	38365	555	firele interioare rupte, traseul pe frontieră
9	138	138	273	274	o "barieră" pe mijlocul tablei, cu o "fantă"

O soluţie backtracking simplu obţine maxim 20 de puncte, iar îmbunătăţit maxim 30 de puncte.

Codul sursă

```
import java.io.*; // test 3 eroare date lm=144 ???
class Paianjen4   // coada circulara (altfel trebuie dimensiune mare !)
{                 // traseu fara recursivitate (altfel depaseste stiva !)
  static final int qdim=200; // dimensiune coada circulara
  static final int sus=1, dreapta=2, jos=4, stanga=8;
  static int[][] p=new int[140][140]; // plasa
  static int[][] c=new int[140][140]; // costul ajungerii in (i,j)
  static int[][] d=new int[140][140]; // directii pentru intoarcere de la musca!

  static int m,n;   // dimensiunile plasei
  static int lp,cp; // pozitie paianjen(lin,col)
  static int lm,cm; // pozitie musca(lin,col)
  static int[] qi=new int[qdim];  // coada pentru i din pozitia (i,j)
  static int[] qj=new int[qdim];  // coada pentru j din pozitia (i,j)
  static int ic, sc;              // inceput/sfarsit coada

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    citescDate();
    matriceCosturi();
    afisSolutia();
    t2=System.currentTimeMillis();
```

```
    System.out.println("TIME = "+(t2-t1)+" millisec ");
  }// main()

  static void citescDate() throws IOException
  {
    int nrSegmenteDeteriorate, k,i,j,l1,c1,l2,c2;
    int i1,i2,j1,j2;

    StreamTokenizer st=new StreamTokenizer(
        new BufferedReader(new FileReader("paianjen.in")));
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); lp=(int)st.nval;
    st.nextToken(); cp=(int)st.nval;
    st.nextToken(); lm=(int)st.nval;
    st.nextToken(); cm=(int)st.nval;
    st.nextToken(); nrSegmenteDeteriorate=(int)st.nval;

    for(i=0;i<m;i++) for(j=0;j<n;j++) p[i][j]=0xF;  // 1111=toate firele !
    for(k=1;k<=nrSegmenteDeteriorate;k++)
    {
      st.nextToken();l1=(int)st.nval;
      st.nextToken();c1=(int)st.nval;
      st.nextToken();l2=(int)st.nval;
      st.nextToken();c2=(int)st.nval;

      i1=min(l1,l2); i2=max(l1,l2);
      j1=min(c1,c2); j2=max(c1,c2);

      if(j1==j2)                 // ruptura verticala
      {
        p[i1][j1]^=jos;          // sau ... p[i1][j1]-=jos; ... !!!
        for(i=i1+1;i<=i2-1;i++)
        {
          p[i][j1]^=jos;         // 0 pe directia jos
          p[i][j1]^=sus;         // 0 pe directia sus
        }
        p[i2][j1]^=sus;
      }
      else
      if(i1==i2)                // ruptura orizontala
      {
        p[i1][j1]^=dreapta;     // 0 pe directia dreapta
        for(j=j1+1;j<=j2-1;j++)
```

```
      {
        p[i1][j]^=dreapta;
        p[i1][j]^=stanga;
      }
      p[i1][j2]^=stanga;     // 0 pe directia stanga
    }
    else System.out.println("Date de intrare ... eronate !");
  }// for k
}//citescDate()

static void matriceCosturi()
{
  int i,j;
  ic=sc=0;                                  // coada vida
  qi[sc]=lp; qj[sc]=cp; sc=(sc+1)%qdim;  // (lp,cp) --> coada !
  c[lp][cp]=1;  // cost 1 pentru pozitie paianjen (pentru marcaj!)
  while(ic!=sc) // coada nevida
  {
    i=qi[ic]; j=qj[ic]; ic=(ic+1)%qdim;
    fill(i,j);
    if(c[lm][cm]!=0) break;  // a ajuns deja la musca !
  }// while
}//matriceCosturi()

static void fill(int i, int j)
{
  int t=c[i][j];  // timp !

  if((i-1>=0)&&(c[i-1][j]==0)&&ok(i,j,sus))  // N
  {
    c[i-1][j]=t+1;
    qi[sc]=i-1;
    qj[sc]=j;
    sc=(sc+1)%qdim;
    d[i-1][j]=jos;
  }

  if((j+1<=n-1)&&(c[i][j+1]==0)&&ok(i,j,dreapta)) // E
  {
    c[i][j+1]=t+1;
    qi[sc]=i;
    qj[sc]=j+1;
    sc=(sc+1)%qdim;
    d[i][j+1]=stanga;
```

```
    }

    if((i+1<=m-1)&&(c[i+1][j]==0)&&ok(i,j,jos)) // S
    {
      c[i+1][j]=t+1;
      qi[sc]=i+1;
      qj[sc]=j;
      sc=(sc+1)%qdim;
      d[i+1][j]=sus;
    }

    if((j-1>=0)&&(c[i][j-1]==0)&&ok(i,j,stanga)) // V
    {
      c[i][j-1]=t+1;
      qi[sc]=i;
      qj[sc]=j-1;
      sc=(sc+1)%qdim;
      d[i][j-1]=dreapta;
    }
  }// fill(...)

  static boolean ok(int i, int j, int dir)
  {
    if((p[i][j]&dir)!=0) return true; else return false;
  }// ok(...)

  static int min(int a, int b)
  {
    if(a<b) return a; else return b;
  }

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }

  static void afisSolutia() throws IOException
  {
    int i,j;
    PrintWriter out=new PrintWriter(
         new BufferedWriter(new FileWriter("paianjen.out")));
    out.println(c[lm][cm]-1);

    i=lm;
```

```
    j=cm;
    while((i!=lp)||(j!=cp))    // folosesc matricea c care nu mai e necesara !
      if(d[i][j]==sus) { c[i-1][j]=jos; i--; }
      else if(d[i][j]==jos) { c[i+1][j]=sus; i++;}
      else if(d[i][j]==dreapta) { c[i][j+1]=stanga; j++; }
      else if(d[i][j]==stanga) { c[i][j-1]=dreapta; j--; }
      else System.out.println("Eroare la traseu ... !");

    i=lp;
    j=cp;
    while((i!=lm)||(j!=cm))
    {
      out.println(i+" "+j);
      if(c[i][j]==sus) i--;
      else if(c[i][j]==jos) i++;
      else if(c[i][j]==dreapta) j++;
      else if(c[i][j]==stanga) j--;
      else System.out.println("Eroare la traseu ... !");
    }
    out.println(i+" "+j);   // pozitia pentru musca !
    out.close();
  }//afisSolutia()
}// class
```

13.2.5 Algoritmul Edmonds-Karp

```
import java.io.*;
class FluxMaxim
{
  static final int WHITE=0, GRAY=1, BLACK=2;
  static final int MAX_NODES=10;
  static final int oo=0x7fffffff;
  static int n, m;                                    // nr noduri, nr arce
  static int[][] c=new int[MAX_NODES+1][MAX_NODES+1];  // capacitati
  static int[][] f=new int [MAX_NODES+1][MAX_NODES+1]; // flux
  static int[] color=new int[MAX_NODES+1];            // pentru bfs
  static int[] p=new int[MAX_NODES+1];     // predecesor (ptr. drum crestere)
  static int ic, sc;                       // inceput coada, sfarsit coada
  static int[] q=new int[MAX_NODES+2];     // coada

  public static void main(String[] args) throws IOException
  {
    int s,t,i,j,k,fluxm;                    // fluxm=flux_maxim
    StreamTokenizer st=new StreamTokenizer(
```

```
                    new BufferedReader(new FileReader("fluxMaxim.in")));
  PrintWriter out=new PrintWriter(
                  new BufferedWriter(new FileWriter("fluxMaxim.out")));
  st.nextToken(); n=(int)st.nval;
  st.nextToken(); m=(int)st.nval;
  st.nextToken(); s=(int)st.nval;
  st.nextToken(); t=(int)st.nval;

  for(k=1;k<=m;k++)
  {
    st.nextToken(); i=(int)st.nval;
    st.nextToken(); j=(int)st.nval;
    st.nextToken(); c[i][j]=(int)st.nval;
  }

  fluxm=fluxMax(s,t);

  System.out.println("\nfluxMax("+s+","+t+") = "+fluxm+" :");
  for(i=1;i<=n;i++)
  {
    for(j=1;j<=n;j++) System.out.print(maxim(f[i][j],0)+"\t");
    System.out.println();
  }
  out.print(fluxm);  out.close();
}// main()

static int fluxMax(int s, int t)
{
  int i, j, u, min, maxf = 0;
  for(i=1; i<=n; i++) for(j=1; j<=n; j++) f[i][j]=0;

  // Cat timp exista drum de crestere a fluxului (in graful rezidual),
  // mareste fluxul pe drumul gasit
  while(bfs(s,t))
  {
    // Determina cantitatea cu care se mareste fluxul
    min=oo;
    for(u=t; p[u]!=-1; u=p[u]) min=minim(min,c[p[u]][u]-f[p[u]][u]);

    // Mareste fluxul pe drumul gasit
    for(u=t; p[u]!=-1; u=p[u])
    {
      f[p[u]][u]+=min;
      f[u][p[u]]-=min; // sau f[u][p[u]]=-f[p[u]][u];
```

```
      }

      maxf += min;
      System.out.print("drum : ");
      drum(t);
      System.out.println(" min="+min+" maxf="+maxf+"\n");
    }// while(...)

    // Nu mai exista drum de crestere a fluxului ==> Gata !!!
    System.out.println("Nu mai exista drum de crestere a fluxului !!!");
    return maxf;
  }// fluxMax(...)

  static boolean bfs(int s, int t) // s=sursa t=destinatie
  {
    // System.out.println("bfs "+s+" "+t+" flux curent :");
    // afism(f);
    int u, v;
    boolean gasitt=false;

    for(u=1; u<=n; u++) { color[u]=WHITE; p[u]=-1; }
    ic=sc=0; // coada vida
    incoada(s);
    p[s]=-1;

    while(ic!=sc)
    {
      u=dincoada();

      // Cauta nodurile albe v adiacente cu nodul u si pune v in coada
      // cand capacitatea reziduala a arcului (u,v) este pozitiva
      for(v=1; v<=n; v++)
        if(color[v]==WHITE && ((c[u][v]-f[u][v])>0))
        {
          incoada(v);
          p[v]=u;
          if(v==t) { gasitt=true; break;}
        }

      if(gasitt) break;
    }//while

    return gasitt;
  }// bfs(...)
```

```
  static void drum(int u)
  {
    if(p[u]!=-1) drum(p[u]);
    System.out.print(u+" ");
  }// drum(...)

  static void afism(int[][] a)
  {
    int i,j;
    for(i=1;i<=n;i++)
    {
      for(j=1;j<=n;j++) System.out.print(a[i][j]+"\t");
      System.out.println();
    }
    // System.out.println();
  }// afism(...)

  static int minim(int x, int y) { return (x<y) ? x : y; }

  static int maxim(int x, int y) { return (x>y) ? x : y; }

  static void incoada(int u)
  {
    q[sc++]=u;
    color[u]=GRAY;
  }

  static int dincoada()
  {
    int u=q[ic++];
    color[u]=BLACK;
    return u;
  }
}// class

/*
6 10 1 6    drum : 1 2 4 6    min=12 maxf=12
1 2 16      drum : 1 3 5 6    min= 4 maxf=16
1 3 13      drum : 1 3 5 4 6  min= 7 maxf=23
2 3  4      Nu mai exista drum de crestere a fluxului !!!
2 4 12      fluxMax(1,6) = 23 :
3 2 10      0   12   11    0    0    0
3 5 14      0    0    0   12    0    0
```

```
4 3  9      0    0    0    0   11    0
4 6 20      0    0    0    0    0   19
5 4  7      0    0    0    7    0    4
5 6  4      0    0    0    0    0    0
*/
```

13.2.6 Cuplaj maxim

```
/* Culori - Descrierea problemei:
Doi elfi au pus pe o masa n patratele si m cerculete. Unul a ales patratelele
si celalalt cerculetele si au desenat pe ele mai multe benzi colorate. Apoi au
inceput sa se joace cu patratelele si cerculetele. Au decis ca un cerculet
poate fi amplasat pe un patratel daca exista cel putin o culoare care apare
pe ambele. Ei doresc sa formeze perechi din care fac parte un cerculet si un
patratel astfel incat sa se obtina cat mai multe perechi.

Date de intrare: Fisierul de intrare de intrare contine pe prima linie
numarul n al patratelelor. Pe fiecare dintre urmatoarele n linii sunt
descrise benzile corespunzatoare unui patratel. Primul numar de pe o astfel
de linie este numarul b al benzilor, iar urmatoarele b numere reprezinta
codurile celor b culori. Urmatoarea linie contine numarul m al cerculetelor.
Pe fiecare dintre urmatoarele m linii sunt descrise benzile corespunzatoare
unui cerculet. Primul numar de pe o astfel de linie este numarul b al
benzilor, iar urmatoarele b numere reprezinta codurile celor b culori.
Numerele de pe o linie vor fi separate prin cte un spatiu. Patratelele si
cerculetele vor fi descrise in ordinea data de numarul lor de ordine.

Date de iesire: Fisierul de iesire trebuie sa contina pe prima linie numarul
k al perechilor formate. Fiecare dintre urmatoarele k va contine cate doua
numere, separate printr-un spatiu, reprezentand numerele de ordine ale unui
patratel, respectiv cerc, care formeaza o pereche.

Restrictii si precizari: numarul patratelelor este cuprins intre 1 si 100;
numarul cerculetelor este cuprins intre 1 si 100; patratelele sunt identificate
prin numere cuprinse intre 1 si n; cerculetele sunt identificate prin numere
cuprinse intre 1 si m; numarul benzilor colorate de pe cerculete si patratele
este cuprins intre 1 si 10; un patratel sau un cerc nu poate face parte din
mai mult decat o pereche; daca exista mai multe solutii trebuie determinata
doar una dintre acestea.
Exemplu
INPUT.TXT OUTPUT.TXT
3         2
1 1       1 1          1  .  1 \
1 2       3 2       /    .      \
```

```
1 3               s=0 - 2  .  2 ---  n+m+1=t
4                     \    .     / /
2 1 2                   3  .  3 / /
1 3                        .     /
2 3 4                      .  4 /
1 4        Timp de executie: 0,5 secunde/test    */

import java.io.*;          // u=0 ==> v=1,2,...,n
class CuplajMaximCulori    // u=1,..,n ==> v=n+1,..,n+m
{                          // u=n+1,..,n+m ==> v=1,2,.,n sau n+m+1(=t)
  static final int WHITE=0, GRAY=1, BLACK=2;
  static final int oo=0x7fffffff;
  static int n, m, ic, sc;
  static int[][] c, f;        // capacitati, flux
  static boolean[][] cp, cc;  // cp = culoare patratel, cc = culoare cerc
  static int[] color, p, q;   // predecesor, coada

  public static void main(String[] args) throws IOException
  {
    citire();
    capacitati();
    scrie(fluxMax(0,n+m+1));
  }// main()

  static void citire() throws IOException
  {
    int i,j,k,nc;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("CuplajMaximCulori.in")));
    st.nextToken(); n=(int)st.nval; cp=new boolean[n+1][11];
    for(i=1;i<=n;i++)
    {
      st.nextToken(); nc=(int)st.nval;
      for(k=1;k<=nc;k++)
      {
        st.nextToken(); j=(int)st.nval;
        cp[i][j]=true;
      }
    }
    st.nextToken(); m=(int)st.nval; cc=new boolean[m+1][11];
    for(i=1;i<=m;i++)
    {
      st.nextToken(); nc=(int)st.nval;
      for(k=1;k<=nc;k++)
```

```
      {
        st.nextToken(); j=(int)st.nval;
        cc[i][j]=true;
      }
    }
  }// citire()

  static void capacitati()
  {
    int i,ic,j,jc;
    c=new int[n+m+2][n+m+2];
    for(i=1;i<=n;i++)
    {
      c[0][i]=1;
      for(ic=1;ic<=10;ic++)
        if(cp[i][ic])
          for(j=1;j<=m;j++)
            if(cc[j][ic]) c[i][j+n]=1;
    }
    for(j=1;j<=m;j++) c[j+n][n+m+1]=1;
  }// capacitati()

  static int fluxMax(int s, int t)
  {
    int i,j,u,min,maxf=0;

    f=new int[n+m+2][n+m+2];
    p=new int[n+m+2];
    q=new int[n+m+2];
    color=new int[n+m+2];

    for(i=0;i<=n+m+1;i++)
      for(j=0;j<=n+m+1;j++) f[i][j]=0;

    while(bfs(s,t))
    {
      min=oo;
      for(u=t;p[u]!=-1;u=p[u]) min=minim(min,c[p[u]][u]-f[p[u]][u]);
      for(u=t;p[u]!=-1;u=p[u])
      {
        f[p[u]][u]+=min;
        f[u][p[u]]-=min; // sau f[u][p[u]]=-f[p[u]][u];
      }
```

```
    maxf+=min;
  }// while(...)

  return maxf;
}// fluxMax(...)

static boolean bfs(int s, int t)
{
  int u, v;
  boolean gasitt=false;
  for(u=0;u<=n+m+1;u++) {color[u]=WHITE; p[u]=-1;}
  ic=sc=0;
  q[sc++]=s; color[s]=GRAY; // s --> coada
  p[s]=-1;
  while(ic!=sc)
  {
    u=q[ic++]; color[u]=BLACK;
    if(u==0)
    {
      for(v=1;v<=n;v++)
      if((color[v]==WHITE)&&((c[u][v]-f[u][v])>0))
      {
        q[sc++]=v; color[v]=GRAY; // incoada(v);
        p[v]=u;
      }
    }
    else if(u<=n)
         {
           for(v=n+1;v<=n+m;v++)
             if((color[v]==WHITE)&&((c[u][v]-f[u][v])>0))
             {
               q[sc++]=v; color[v]=GRAY; // incoada(v);
               p[v]=u;
             }
         }
         else
         {
           for(v=n+m+1;v>=1;v--)
             if((color[v]==WHITE)&&((c[u][v]-f[u][v])>0))
             {
               q[sc++]=v; color[v]=GRAY; // incoada(v);
               p[v]=u;
               if(v==t) {gasitt=true; break;}
             }
```

```
            if(gasitt) break; // din while !
          }
    }// while()

    return gasitt;
  }// bfs()


  static int minim(int x, int y) { return (x<y) ? x : y; }

  static int maxim(int x, int y) { return (x>y) ? x : y; }

  static void scrie(int fluxm) throws IOException
  {
    int i,j;
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("CuplajMaximCulori.out")));
    out.println(fluxm);
    for (i=1;i<=n;i++)
      if(f[0][i]>0)
        for(j=1;j<=m;j++)
          if(f[i][j+n]>0)
          {
            out.println(i+" "+j);
            break;
          }
    out.close();
  }// scrie(...)
}// class
```

Capitolul 14

Metoda optimului local - greedy

14.1 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 C (*candidaţi* la construirea *soluţiei* problemei), se cere să se determine o submulţime S, (*soluţia* problemei) care *îndeplineşte* anumite *condiţii*. Deoarece este posibil să existe mai multe soluţii se va alege soluţia care *maximizează* sau *minimizează* o anumită *funcţie obiectiv*.

O problemă poate fi rezolvată prin tehnica (metoda) Greedy dacă îndeplineşte proprietatea: dacă S este o *soluţie*, iar S' este inclusă în S, atunci şi S' este o *soluţie*.

Pornind de la această condiţie, iniţial se presupune că S este mulţimea vidă şi se adaugă succesiv elemente din C în S, ajungând la un *optim local*. 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 cu succes.

Există următoarele variante ale metodei Greedy:

1. Se pleacă de la soluţia vidă pentru multimea S şi se ia pe rând câte un element din mulţimea C. Dacă elementul ales îndeplineşte *condiţia de optim local*, el este introdus în mulţimea S.

2. Se ordonează elementele mulţimii C şi se verifică dacă un element îndeplineşte condiţia de apartenenţă la mulţimea S.

14.2 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 încât 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 selectaţi 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 rămâne de acum încolo în ea. De fiecare dată când lărgim mulţimea candidaţilor selectaţi, verificăm dacă această mulţime constituie o *soluţie posibilă* a problemei. Dacă algoritmul *greedy* funcţionează corect, prima soluţie găsită va fi considerată *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"

14.3 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.

14.3.1 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*()
    $G_p \leftarrow G$
    for $i = 1, n$
        if $G_p > g_i$
            then $G_p \leftarrow G_p - g_i$

else
$x_i \leftarrow \frac{G_p}{g_i}$
for $j = i+1, n$
$x_j \leftarrow 0$

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.

14.3.2 Problema plasării textelor pe o bandă

Să presupunem că trebuie să plasăm n texte $T_1, T_2, ..., T_n$, de lungimi date $L_1, L_2, ..., L_n$, pe o singură bandă suficient de lungă. Atunci când este necesară citirea unui text sunt citite toate textele situate înaintea lui pe bandă.

Modalitatea de plasare a celor n texte pe bandă corespunde unei permutări p a mulţimii $\{1, 2, ..., n\}$, textele fiind aşezate pe bandă în ordinea $T_{p(1)}T_{p(2)}...T_{p(n)}$.

Într-o astfel de aranjare a textelor pe bandă, timpul mediu de citire a unui text este:

$$f(p) = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L_{p(i)}$$

Se doreşte determinarea unei permutări care să asigure o valoare *minimă* a timpului mediu de citire. Rezolvarea problemei este foarte simplă:

- **se sortează crescător textele în funcţie de lungimea lor şi**
- **se plasează pe bandă în ordinea dată de sortare.**

Următoarea propoziţie ([25] pagina 99) ne permite să fim siguri că în acest mod ajungem la o plasare optimă.

Propoziţia 2 *Dacă $L_1 \leq L_2 \leq ... \leq L_n$ atunci plasarea textelor corespunzătoare permutarii identice este optimă.*

Demonstraţie: Fie $p = (p_1, p_2, ..., p_n)$ o plasare optimă. Dacă există $i < j$ cu $L_{p(i)} \geq L_{p(j)}$ atunci considerăm permutarea p' obţinută din p prin permutarea elementelor de pe poziţiile i şi j. Atunci

$$f(p') - f(p) = (n - j + 1)(L_{p(i)} - L_{p(j)}) + (n - i + 1)(L_{p(j)} - L_{p(i)})$$

deci

$$f(p') - f(p) = (L_{p(j)} - L_{p(i)})(j - i) \leq 0.$$

Cum p este o plasare optimă şi $f(p') \leq f(p)$ rezultă că $f(p') = f(p)$ deci şi p' este o plasare optimă. Aplicând de un număr finit de ori acest raţionament, trecem de la o permutare optimă la altă permutare optimă până ajungem la permutarea identică. Rezultă că permutarea identică corespunde unei plasări optime.

14.3.3 Problema plasării textelor pe m benzi

Să presupunem că trebuie să plasăm n texte $T_1, T_2, ..., T_n$, de lungimi date $L_1, L_2, ..., L_n$, pe m benzi suficient de lungi. Atunci când este necesară citirea unui text sunt citite toate textele situate înaintea lui pe banda respectivă.

Se doreşte determinarea unei plasări a celor n texte pe cele m benzi care să asigure o valoare *minimă* a timpului mediu de citire. Rezolvarea problemei este foarte simplă:

- **se sortează crescător textele în funcţie de lungimea lor şi**
- **plasarea textelor pe benzi se face în ordinea dată de sortare, începând cu primul text (cu cea mai mică lungime) care se plasează pe prima bandă, iar mai departe**
- **fiecare text se plasează pe banda următoare celei pe care a fost plasat textul anterior.**

Presupunând că $L_1 \leq L_2 \leq ... \leq L_n$, se poate observa că textul T_i va fi plasat pe banda $i - \left\lfloor \frac{i-1}{m} \right\rfloor \cdot m$ în continuarea celor aflate deja pe această bandă iar după textul T_i, pe aceeaşi bandă cu el, vor fi plasate textele $i+m, i+2m, ...$, deci încă $\left\lfloor \frac{n-i}{m} \right\rfloor$ texte (demonstraţiile se pot consulta, de exemplu, în [25]).

14.3.4 Maximizarea unei sume de produse

Se dau mulţimile de numere întregi $A = \{a_1, a_2, ..., a_n\}$ şi $B = \{b_1, b_2, ..., b_m\}$, unde $n \leq m$. Să se determine o submulţime $B' = \{x_1, x_2, ..., x_n\}$ a lui B astfel încât valoarea expresiei $E = a_1 \cdot x_1 + a_2 \cdot x_2 + a_n \cdot x_n$ să fie maximă.

Vom sorta crescător elementele celor două mulţimi. Putem presupune acum că $a_1 \leq a_2 \leq ... \leq a_n$ şi $b_1 \leq b_2 \leq ... \leq b_m$.

Dacă toate elementele din A sunt pozitive vom lua $x_n = b_m$, $x_{n-1} = b_{m-1}$, şi aşa mai departe.

Dacă toate elementele din A sunt negative vom lua $x_1 = b_1$, $x_2 = b_2$, şi aşa mai departe.

Dacă primele n_1 elemente din A sunt strict negative şi ultimele n_2 elemente din A sunt pozitive sau nule ($n_1 + n_2 = n$) atunci vom lua $x_1 = b_1$, $x_2 = b_2$, ..., $x_{n_1} = b_{n_1}$ şi $x_n = b_m$, $x_{n-1} = b_{m-1}$, ..., $x_{n-n_2+1} = b_{m-n_2+1}$.

14.3.5 Problema staţiilor

Se consideră o mulţime de numere naturale $A = \{a_1, a_2, ..., a_n\}$ care reprezintă coordonatele a n staţii pe axa reală. Vom presupune $a_1 < a_2 < ... < a_n$. Să se

determine o submulţime cu număr maxim de staţii cu proprietatea că distanţa dintre două staţii alăturate este cel putin d (o distanţă dată).

Rezolvarea problemei este următoarea:

- se alege staţia 1,
- se parcurge şirul staţiilor şi se alege prima staţie întâlnită care este la distanţă cel puţin d faţă de staţia aleasă anterior; se repetă acest pas până când s-au verificat toate staţiile.

Propoziţia 3 *Există o soluţie optimă care conţine staţia 1.*

Demonstraţie: Fie $B = \{a_{i_1}, a_{i_2}, ..., a_{i_m}\}$ o soluţie a problemei care nu conţine staţia 1 (deci $a_{i_1} > a_1$). Evident $|a_{i_2} - a_{i_1}| \geq d$. Staţia i_1 se poate înlocui cu staţia 1 pentru că $|a_{i_2} - a_1| = |a_{i_2} - a_{i_1} + a_{i_1} - a_1| = |a_{i_2} - a_{i_1}| + |a_{i_1} - a_1| > d$.

După selectarea staţie 1 se pot selecta (pentru obţinerea unei soluţii optime) numai staţii situate la distanţe cel puţin d fată de staţia 1. Pentru aceste staţii repetăm strategia sugerată de propoziţia anterioară.

14.3.6 Problema cutiilor

Se doreşte obţinerea unei configuraţii de n numere plasate pe $n+1$ poziţii (o poziţie fiind liberă) dintr-o configuraţie iniţială dată în care există o poziţie liberă şi cele n numere plasate în altă ordine. O mutare se poate face dintr-o anumită poziţie numai în poziţia liberă.

Prezentăm algoritmul de rezolvare pe baza următorului exemplu:

	1	2	3	4	5	6	7
iniţial →	3	1		2	6	5	4
final →	1	2	3		4	5	6
rezolvat →						×	

Vom proceda astfel: cutia goală din configuraţia iniţială se află pe poziţia 3 dar pe această poziţie, în configuraţia finală, se află numărul 3; căutăm numărul 3 din configuraţia iniţială (îl găsim pe poziţia 1) şi îl mutăm pe poziţia cutiei goale; acum, cutia goală se află pe poziţia 1; vom repeta acest raţionament până când poziţiile cutiilor goale, în cele două configuraţii, coincid.

	1	2	3	4	5	6	7
modificat →		1	3	2	6	5	4
final →	1	2	3		4	5	6
rezolvat →			×			×	

	1	2	3	4	5	6	7
modificat →		1	3	2	6	5	4
final →	1	2	3		4	5	6
rezolvat →			×			×	

	1	2	3	4	5	6	7
modificat →	1	2	3		6	5	4
final →	1	2	3		4	5	6
rezolvat →	×	×	×			×	

Acum vom căuta o cutie nerezolvată şi vom muta numărul din acea cutie în cutia goală.

	1	2	3	4	5	6	7
modificat →	1	2	3	6		5	4
final →	1	2	3		4	5	6
rezolvat →	×	×	×			×	

Repetăm raţionamentul prezentat la început şi vom continua până când toate numerele ajung pe poziţiile din configuraţia finală.

	1	2	3	4	5	6	7
modificat →	1	2	3	6	4	5	
final →	1	2	3		4	5	6
rezolvat →	×	×	×		×	×	

	1	2	3	4	5	6	7
modificat →	1	2	3		4	5	6
final →	1	2	3		4	5	6
rezolvat →	×	×	×		×	×	×

14.3.7 Problema subşirurilor

Să se descompună un şir de n numere întregi în subşiruri strict crescătoare astfel încât numerele lor de ordine din şirul iniţial să fie ordonate crescător în subşirurile formate şi numărul subşirurilor să fie minim.

Metota de rezolvare este următoarea: se parcurg elementele şirului iniţial unul după altul şi pentru fiecare element x_i se caută un subşir existent la care x_i se poate adăuga la sfârşit; dacă nu există un astfel de subşir se creează un subşir nou care va conţine ca prim element pe x_i; dacă există mai multe subşiruri la care se poate adăuga x_i, se va alege acela care are cel mai mare ultim element.

Observaţie: Ultimele elemente din fiecare subşir (care sunt şi elemente maxime din aceste subşiruri) formează un şir descrescător. Acest fapt permite utilizarea **căutării binare** pentru determinarea subşirului potrivit pentru elementul x_i atunci când prelucrăm acest element.

14.3.8 Problema intervalelor disjuncte

Se consideră n intervale închise pe axa reală $[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]$. Se cere selectarea unor intervale disjuncte astfel încât numărul acestora să fie maxim.

Metoda de rezolvare este următoarea: se sortează intervalele crescător după capătul din dreapta; se selectează primul interval; se parcurg intervalele, unul

după altul, până se găseşte un interval $[a_i, b_i]$ disjunct cu ultimul interval ales şi se adaugă acest interval la soluţie, el devenind astfel "ultimul interval ales"; acest procedeu continuă până când nu mai rămân intervale de analizat.

Propoziţia 4 *Există o soluţie optimă care conţine primul interval după sortare.*

14.3.9 Problema alegerii taxelor

Se dau două şiruri cu câte $2n$ numere întregi fiecare, $x = (x_1, x_2, ..., x_{2n})$ şi $y = (y_1, y_2, ..., y_{2n})$ reprezentând taxe. Să se determine şirul $z = (z_1, z_2, ..., z_{2n})$, unde $z_i \in \{x_i, y_i\}$ $(1 \leq i \leq 2n)$, astfel încât suma tuturor elementelor din şirul z să fie minimă şi acest şir să conţină exact n elemente din şirul x şi n elemente din şirul y.

Metoda de rezolvare este următoarea: se construieşte şirul $t = x - y$ (în care $t_i = x_i - y_i$) şi se sortează crescător; se aleg din şirul x elementele corespunzătoare primelor n elemente din şirul t sortat iar celelalte n elemente se iau din şirul y.

14.3.10 Problema acoperirii intervalelor

Se consideră n intervale închise $[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]$. Să se determine o mulţime cu număr minim de alemente $C = \{c_1, c_2, ..., c_m\}$ care să "acopere" toate cele n intervale date (spunem că c_i "acoperă" intervalul $[a_k, b_k]$ dacă $c_i \in [a_k, b_k]$).

Metoda de rezolvare este următoarea: se sortează intervalele crescător după capătul din dreapta. Presupunem că $b_1 \leq b_2 \leq ... \leq b_n$. Primul punct ales este $c_1 = b_1$. Parcurgem intervalele până cand găsim un interval $[a_i, b_i]$ cu $a_i > c_1$ şi alegem $c_2 = b_i$. Parcurgem mai departe intervalele până cand găsim un interval $[a_j, b_j]$ cu $a_j > c_2$ şi alegem $c_3 = b_j$. Repetăm acest procedeu până când terminăm de parcurs toate intervalele.

14.3.11 Algoritmul lui Prim

Determinarea arborelui minim de acoperire pentru un graf neorientat.

Alg prim de ordinul $O(n^3)$

```
import java.io.*;   // arbore minim de acoperire: algoritmul lui Prim
class Prim   // O(n^3)
{
  static final int oo=0x7fffffff;
  static int n,m;
  static int[][] cost;
  static boolean[] esteInArbore;
```

```
  static int[] p;                // predecesor in arbore

  public static void main(String[]args) throws IOException
  {
    int nods=3;                    // nod start
    int i, j, k, costArbore=0,min,imin=0,jmin=0;

    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("prim.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("prim.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    cost=new int[n+1][n+1];
    esteInArbore=new boolean [n+1];
    p=new int[n+1];

    for(i=1;i<=n;i++) for(j=1;j<=n;j++) cost[i][j]=oo;
    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost[i][j]=cost[j][i]=(int)st.nval;
    }

    esteInArbore[nods]=true;
    for(k=1;k<=n-1;k++)     // sunt exact n-1 muchii in arbore !!! O(n)
    {
      min=oo;
      for(i=1;i<=n;i++)                // O(n)
      {
        if(!esteInArbore[i]) continue;
        for(j=1;j<=n;j++)                 // O(n)
        {
          if(esteInArbore[j]) continue;
          if(min>cost[i][j]) { min=cost[i][j]; imin=i; jmin=j; }
        }//for j
      }//for i

      esteInArbore[jmin]=true;
      p[jmin]=imin;
      costArbore+=min;
```

```
    }//for k

    for(k=1;k<=n;k++) if(p[k]!=0) out.println(k+" "+p[k]);
    out.println("cost="+costArbore);
    out.close();
  }//main
}//class

/*
6 7           1 3
1 2 3         2 3
1 3 1         4 3
2 3 2         5 4
3 4 1         6 4
4 5 1         cost=7
5 6 3
4 6 2
*/
```

Alg Prim cu distanţe

```
import java.io.*;    // arbore minim de acoperire: algoritmul lui Prim
class PrimDist       // O(n^2)
{
  static final int oo=0x7fffffff;
  static int n,m;
  static int[][] cost;
  static boolean[] esteInArbore;
  static int[] p;                 // predecesor in arbore
  static int[] d;                 // distante de la nod catre arbore

  public static void main(String[]args) throws IOException
  {
    int nodStart=3;              // nod start
    int i, j, k, costArbore=0,min,jmin=0;

    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("prim.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("prim.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
```

```
cost=new int[n+1][n+1];
esteInArbore=new boolean [n+1];
p=new int[n+1];
d=new int[n+1];

for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
    cost[i][j]=oo;
for(i=1;i<=n;i++) d[i]=oo;

for(k=1;k<=m;k++)
{
  st.nextToken(); i=(int)st.nval;
  st.nextToken(); j=(int)st.nval;
  st.nextToken(); cost[i][j]=cost[j][i]=(int)st.nval;
}

d[nodStart]=0;

for(k=1;k<=n;k++)         //  O(n)
{
  min=oo;
  for(j=1;j<=n;j++)       // O(n)
  {
    if(esteInArbore[j]) continue;
    if(min>d[j]) { min=d[j]; jmin=j; }
  }//for j

  esteInArbore[jmin]=true;
  d[jmin]=0;
  costArbore+=min;

  for(j=1;j<=n;j++)       // actualizez distantele nodurilor O(n)
  {
    if(esteInArbore[j]) continue;    // j este deja in arbore
    if(cost[jmin][j]<oo)             // am muchia (jmin,j)
    if(d[jmin]+cost[jmin][j]<d[j])
    {
      d[j]=d[jmin]+cost[jmin][j];
      p[j]=jmin;
    }
  }
}//for k
```

```
    for(k=1;k<=n;k++) if(p[k]!=0) out.println(k+" "+p[k]);
    out.println("cost="+costArbore);
    out.close();
  }//main
}//class

/*
6 7          1 3
1 2 3        2 3
1 3 1        4 3
2 3 2        5 4
3 4 1        6 4
4 5 1        cost=7
5 6 3
4 6 2
*/
```

Alg Prim cu heap

```
import java.io.*;   // arbore minim de acoperire: algoritmul lui Prim
class PrimHeap      // folosesc distantele catre arbore
{                   // pastrate in MinHeap ==> O(n log n) ==> OK !!!
  static final int oo=0x7fffffff;
  static int n,m;
  static int[][] w;      // matricea costurilor
  static int[] d;        // distante de la nod catre arbore
  static int[] p;        // predecesorul nodului in arbore
  static int[] hd;       // hd[i]=distanta din pozitia i din heap
  static int[] hnod;     // hnod[i]= nodul din pozitia i din heap
  static int[] pozh;     // pozh[i]=pozitia din heap a nodului i

  public static void main(String[]args) throws IOException
  {
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("prim.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("prim.out")));
    int i,j,k,cost,costa=0,nods;

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); nods=(int)st.nval;

    w=new int[n+1][n+1];
```

```
    for(i=1;i<=n;i++) for(j=1;j<=n;j++) w[i][j]=oo;

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost=(int)st.nval;
      w[j][i]=w[i][j]=cost;
    }

    prim(nods);

    for(i=1;i<=n;i++)          // afisez muchiile din arbore
      if(i!=nods) {out.println(p[i]+" "+i);costa+=w[p[i]][i];}
    out.println("costa="+costa);
    out.close();
  }//main

  static void prim(int nods)
  {
    int u,v,q,aux;

    d=new int [n+1];
    p=new int [n+1];
    hd=new int[n+1];
    hnod=new int[n+1];
    pozh=new int[n+1];

    for(u=1;u<=n;u++)
    {
      hnod[u]=pozh[u]=u;
      hd[u]=d[u]=oo;
      p[u]=0;
    }

    q=n;              // q = noduri nefinalizate = dimensiune heap
    d[nods]=0;
    hd[pozh[nods]]=0;

    urcInHeap(pozh[nods]);
    while(q>0)           // la fiecare pas adaug un varf in arbore
    {
      u=extragMin(q);
      if(u==-1) { System.out.println("graf neconex !!!"); break; }
```

```
    q--;
    for(v=1;v<=q;v++)         // noduri nefinalizate = in heap 1..q
      if(w[u][hnod[v]]<oo)    // cost finit ==> exista arc (u,v)
        relax(u,hnod[v]);     // relax si refac heap
  }
}//prim(...)

static void relax(int u,int v)
{
  if(w[u][v]<d[v])
  {
    d[v]=w[u][v];
    p[v]=u;
    hd[pozh[v]]=d[v];
    urcInHeap(pozh[v]);
  }
}//relax(...)

static int extragMin(int q)   // in heap 1..q
{
  // returnez valoarea minima (din varf!) si refac heap in jos
  // aducand ultimul in varf si coborand !

  int aux,fiu1,fiu2,fiu,tata,nod;

  aux=hd[1]; hd[1]=hd[q]; hd[q]=aux;           // 1 <---> q

  aux=hnod[1]; hnod[1]=hnod[q]; hnod[q]=aux;

  pozh[hnod[q]]=q;
  pozh[hnod[1]]=1;

  tata=1;
  fiu1=2*tata;
  fiu2=2*tata+1;

  while(fiu1<=q-1)          //refac heap de sus in jos pana la q-1
  {
    fiu=fiu1;
    if(fiu2<=q-1)
      if(hd[fiu2]<hd[fiu1])
        fiu=fiu2;

    if(hd[tata]<=hd[fiu])
```

```
        break;

      pozh[hnod[fiu]]=tata;
      pozh[hnod[tata]]=fiu;

      aux=hd[fiu]; hd[fiu]=hd[tata]; hd[tata]=aux;

      aux=hnod[fiu]; hnod[fiu]=hnod[tata]; hnod[tata]=aux;

      tata=fiu;
      fiu1=2*tata;
      fiu2=2*tata+1;
    }

    return hnod[q];    // hnod[1] a ajuns deja (!) in hnod[q]
  }// extragMin(...)


  static void urcInHeap(int nodh)
  {
    int aux,fiu,tata,nod;

    nod=hnod[nodh];
    fiu=nodh;
    tata=fiu/2;
    while((tata>0)&&(hd[fiu]<hd[tata]))
    {
      pozh[hnod[tata]]=fiu;
      hnod[fiu]=hnod[tata];

      aux=hd[fiu]; hd[fiu]=hd[tata]; hd[tata]=aux;

      fiu=tata;
      tata=fiu/2;
    }
    pozh[nod]=fiu;
    hnod[fiu]=nod;
  }
}//class

/*
6 7 3
1 2 3        3 1
1 3 1        3 2
```

```
2 3 2        3 4
3 4 1        4 5
4 5 1        4 6
5 6 3        costa=7
4 6 2
*/
```

14.3.12 Algoritmul lui Kruskal

```
import java.io.*; // Arbore minim de acoperire : Kruskal
class Kruskal
{
  static int n,m,cost=0;
  static int[] x,y,z,et;

  public static void main(String[] args) throws IOException
  {
    int k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("kruskal.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("kruskal.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    x=new int[m+1];
    y=new int[m+1];
    z=new int[m+1];
    et=new int[n+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
      st.nextToken(); z[k]=(int)st.nval;
    }

    kruskal();

    System.out.println("cost="+cost);
    out.println(cost);
    out.close();
  }//main
```

```
static void kruskal()
{
  int nm=0,k,etg1,etg2;

  for(k=1;k<=n;k++) et[k]=k;

  qsort(1,m,z);

  for(k=1;k<=m;k++)
  {
    if(et[x[k]]!=et[y[k]])
    {
      nm++;
      cost+=z[k];
      System.out.println(x[k]+" "+y[k]);
      etg1=et[x[k]];
      etg2=et[y[k]];
      for(int i=1;i<=n;i++)
        if(et[i]==etg2) et[i]=etg1;
    }
    if(nm==n-1)break;
  }
}//kruskal

static void qsort(int p, int u, int []x)
{
  int k=poz(p,u,x);
  if(p<k-1) qsort(p,k-1,x);
  if(k+1<u) qsort(k+1,u,x);
}

static void invers(int i, int j, int x[])
{
  int aux;
  aux=x[i]; x[i]=x[j]; x[j]=aux;
}

static int poz(int p, int u, int z[])
{
  int k,i,j;
  i=p; j=u;
  while(i<j)
  {
```

```
      while((z[i]<=z[j])&&(i<j)) i++;
      while((z[i]<=z[j])&&(i<j)) j--;
       if(i<j) { invers(i,j,z); invers(i,j,x); invers(i,j,y); }
    }
    return i; //i==j
  }//poz
}//class
```

14.3.13 Algoritmul lui Dijkstra

Alg Dijkstra cu distanţe, în graf neorientat

```
import java.io.*;  // drumuri minime de la nodSursa
class Dijkstra     // O(n^2)
{
  static final int oo=0x7fffffff;
  static int n,m;
  static int[][] cost;
  static boolean[] esteFinalizat;
  static int[] p;          // predecesor in drum
  static int[] d;          // distante de la nod catre nodSursa

  public static void main(String[]args) throws IOException
  {
    int nodSursa=1;        // nod start
    int i, j, k, min,jmin=0;

    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("dijkstraNeorientat.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("dijkstraNeorientat.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    cost=new int[n+1][n+1];
    esteFinalizat=new boolean [n+1];
    p=new int[n+1];
    d=new int[n+1];

    for(i=1;i<=n;i++) for(j=1;j<=n;j++) cost[i][j]=oo;
    for(i=1;i<=n;i++) d[i]=oo;

    for(k=1;k<=m;k++)
```

```
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost[i][j]=cost[j][i]=(int)st.nval;
    }

    d[nodSursa]=0;

    for(k=1;k<=n;k++)     //  O(n)
    {
      min=oo;
      for(j=1;j<=n;j++)  // O(n)
      {
        if(esteFinalizat[j]) continue;
        if(min>d[j])  { min=d[j]; jmin=j; }
      }//for j
      esteFinalizat[jmin]=true;
      for(j=1;j<=n;j++) // actualizez distantele nodurilor // O(n)
      {
        if(esteFinalizat[j]) continue;  // j este deja in arbore
        if(cost[jmin][j]<oo)            // am muchia (jmin,j)
          if(d[jmin]+cost[jmin][j]<d[j])
          {
            d[j]=d[jmin]+cost[jmin][j];
            p[j]=jmin;
          }
      }
    }//for k
    for(k=1;k<=n;k++)
    {
      System.out.print(nodSursa+"-->"+k+" dist="+d[k]+" drum: ");
      drum(k);
      System.out.println();
    }
    out.close();
  }//main
  static void drum(int k) // s --> ... --> k
  {
    if(p[k]!=0) drum(p[k]);
    System.out.print(k+" ");
  }
}//class
/*
6 7             1-->1 dist=0 drum: 1
```

```
1 2 4           1-->2 dist=3 drum: 1 3 2
1 3 1           1-->3 dist=1 drum: 1 3
2 3 2           1-->4 dist=2 drum: 1 3 4
3 4 1           1-->5 dist=3 drum: 1 3 4 5
5 4 1           1-->6 dist=4 drum: 1 3 4 6
5 6 3
4 6 2
*/
```

Alg Dijkstra cu heap, în graf neorientat

```
import java.io.*;              // distante minime de la nodSursa
class DijkstraNeorientatHeap   // pastrate in MinHeap ==> O(n log n) ==> OK !!!
{
  static final int oo=0x7fffffff;
  static int n,m;
  static int[][]w;     // matricea costurilor
  static int[]d;       // distante de la nod catre arbore
  static int[]p;       // predecesorul nodului in arbore
  static int[] hd;     // hd[i]=distanta din pozitia i din heap
  static int[] hnod;   // hnod[i]= nodul din pozitia i din heap
  static int[] pozh;   // pozh[i]=pozitia din heap a nodului i

  public static void main(String[]args) throws IOException
  {
    int i,j,k,cost,costa=0,nodSursa;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("dijkstraNeorientat.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("dijkstraNeorientat.out")));
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    w=new int[n+1][n+1];
    for(i=1;i<=n;i++) for(j=1;j<=n;j++) w[i][j]=oo;
    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost=(int)st.nval;
      w[j][i]=w[i][j]=cost;
    }

    nodSursa=1;
    dijkstra(nodSursa);
```

```
    for(k=1;k<=n;k++)
    {
      System.out.print(nodSursa+"-->"+k+" dist="+d[k]+" drum: ");
      drum(k);
      System.out.println();
    }
    out.close();
  }//main

  static void dijkstra(int nods)
  {
    int u,v,q,aux;

    d=new int [n+1];
    p=new int [n+1];
    hd=new int[n+1];
    hnod=new int[n+1];
    pozh=new int[n+1];

    for(u=1;u<=n;u++)
    {
      hnod[u]=pozh[u]=u;
      hd[u]=d[u]=oo;
      p[u]=0;
    }
    q=n;          // q = noduri nefinalizate = dimensiune heap
    d[nods]=0;
    hd[pozh[nods]]=0;

    urcInHeap(pozh[nods]);

    while(q>0)     // la fiecare pas adaug un varf in arbore
    {
      u=extragMin(q);
      if(u==-1) { System.out.println("graf neconex !!!"); break; }
      q--;
      for(v=1;v<=q;v++)           // noduri nefinalizate = in heap 1..q
        if(w[u][hnod[v]]<oo)      // cost finit ==> exista arc (u,v)
          relax(u,hnod[v]);       // relax si refac heap
    }
  }// dijkstra()

  static void relax(int u,int v)
```

```
{
  if(d[u]+w[u][v]<d[v])
  {
    d[v]=d[u]+w[u][v];
    p[v]=u;
    hd[pozh[v]]=d[v];
    urcInHeap(pozh[v]);
  }
}// relax(...)

static int extragMin(int q) // in heap 1..q
{
  // returnez valoarea minima (din varf!) si refac heap in jos
  int aux,fiu1,fiu2,fiu,tata,nod;

  aux=hd[1]; hd[1]=hd[q]; hd[q]=aux; // 1 <---> q
  aux=hnod[1]; hnod[1]=hnod[q]; hnod[q]=aux;
  pozh[hnod[q]]=q;
  pozh[hnod[1]]=1;

  tata=1;
  fiu1=2*tata;
  fiu2=2*tata+1;

  while(fiu1<=q-1) //refac heap de sus in jos pana la q-1
  {
    fiu=fiu1;
    if(fiu2<=q-1)
      if(hd[fiu2]<hd[fiu1]) fiu=fiu2;
    if(hd[tata]<=hd[fiu]) break;

    pozh[hnod[fiu]]=tata;
    pozh[hnod[tata]]=fiu;

    aux=hd[fiu]; hd[fiu]=hd[tata]; hd[tata]=aux;
    aux=hnod[fiu]; hnod[fiu]=hnod[tata]; hnod[tata]=aux;

    tata=fiu;
    fiu1=2*tata;
    fiu2=2*tata+1;
  }

  return hnod[q]; // hnod[1] a ajuns deja (!) in hnod[q]
} // extragMin(...)
```

```
  static void urcInHeap(int nodh)
  {
    int aux,fiu,tata,nod;

    nod=hnod[nodh];
    fiu=nodh;
    tata=fiu/2;

    while((tata>0)&&(hd[fiu]<hd[tata]))
    {
      pozh[hnod[tata]]=fiu;
      hnod[fiu]=hnod[tata];
      aux=hd[fiu]; hd[fiu]=hd[tata]; hd[tata]=aux;
      fiu=tata;
      tata=fiu/2;
    }
    pozh[nod]=fiu;
    hnod[fiu]=nod;
  } // urcInHeap(...)

  static void drum(int k) // s --> ... --> k
  {
    if(p[k]!=0) drum(p[k]);
    System.out.print(k+" ");
  }
}//class

/*
6 7           1-->1 dist=0 drum: 1
1 2 4         1-->2 dist=3 drum: 1 3 2
1 3 1         1-->3 dist=1 drum: 1 3
2 3 2         1-->4 dist=2 drum: 1 3 4
3 4 1         1-->5 dist=3 drum: 1 3 4 5
5 4 1         1-->6 dist=4 drum: 1 3 4 6
5 6 3
4 6 2
*/
```

Alg Dijkstra cu distanţe, în graf orientat

```
import java.io.*; // drumuri minime de la nodSursa
class Dijkstra    // O(n^2)
{
```

```
static final int oo=0x7fffffff;
static int n,m;
static int[][] cost;
static boolean[] esteFinalizat;
static int[] p;                 // predecesor in drum
static int[] d;                 // distante de la nod catre nodSursa

public static void main(String[]args) throws IOException
{
  int nodSursa=1;              // nod start
  int i, j, k, min,jmin=0;

  StreamTokenizer st=new StreamTokenizer(
                     new BufferedReader(new FileReader("dijkstraOrientat.in")));
  PrintWriter out=new PrintWriter(
                 new BufferedWriter(new FileWriter("dijkstraOrientat.out")));

  st.nextToken(); n=(int)st.nval;
  st.nextToken(); m=(int)st.nval;

  cost=new int[n+1][n+1];
  esteFinalizat=new boolean [n+1];
  p=new int[n+1];
  d=new int[n+1];

  for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
      cost[i][j]=oo;
  for(i=1;i<=n;i++) d[i]=oo;

  for(k=1;k<=m;k++)
  {
    st.nextToken(); i=(int)st.nval;
    st.nextToken(); j=(int)st.nval;
    st.nextToken(); cost[i][j]=(int)st.nval;
  }

  d[nodSursa]=0;

  for(k=1;k<=n;k++) //  O(n)
  {
    min=oo;
    for(j=1;j<=n;j++) // O(n)
    {
```

```
        if(esteFinalizat[j]) continue;
        if(min>d[j]) { min=d[j]; jmin=j; }
      }//for j
      esteFinalizat[jmin]=true;
      for(j=1;j<=n;j++) // actualizez distantele nodurilor // O(n)
      {
        if(esteFinalizat[j]) continue; // j este deja in arbore
        if(cost[jmin][j]<oo)           // am muchia (jmin,j)
          if(d[jmin]+cost[jmin][j]<d[j])
            {
              d[j]=d[jmin]+cost[jmin][j];
              p[j]=jmin;
            }
      }
    }//for k
    for(k=1;k<=n;k++)
    {
      System.out.print(nodSursa+"-->"+k+" dist="+d[k]+" drum: ");
      if(d[k]<oo) drum(k); else System.out.print("Nu exista drum!");
      System.out.println();
    }
    out.close();
  }//main

  static void drum(int k) // s --> ... --> k
  {
    if(p[k]!=0) drum(p[k]);
    System.out.print(k+" ");
  }
}//class
/*
6 7             1-->1 dist=0 drum: 1
1 2 4           1-->2 dist=4 drum: 1 2
1 3 1           1-->3 dist=1 drum: 1 3
2 3 2           1-->4 dist=2 drum: 1 3 4
3 4 1           1-->5 dist=2147483647 drum: Nu exista drum!
5 4 1           1-->6 dist=4 drum: 1 3 4 6
5 6 3
4 6 2   */
```

Alg Dijkstra cu heap, în graf orientat

```
import java.io.*;           // distante minime de la nodSursa
class DijkstraOrientatHeap  // pastrate in MinHeap ==> O(n log n) ==> OK !!!
```

```
{
  static final int oo=0x7fffffff;
  static int n,m;

  static int[][]w;    // matricea costurilor
  static int[]d;      // distante de la nod catre arbore
  static int[]p;      // predecesorul nodului in arbore

  static int[] hd;    // hd[i]=distanta din pozitia i din heap
  static int[] hnod;  // hnod[i]= nodul din pozitia i din heap
  static int[] pozh;  // pozh[i]=pozitia din heap a nodului i

  public static void main(String[]args) throws IOException
  {
    int i,j,k,cost,costa=0,nodSursa;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("dijkstraOrientat.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("dijkstraOrientat.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    w=new int[n+1][n+1];

    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
        w[i][j]=oo;

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost=(int)st.nval;
      w[i][j]=cost;
    }

    nodSursa=1;
    dijkstra(nodSursa);

    for(k=1;k<=n;k++)
    {
      if(d[k]<oo)
      {
```

```
        System.out.print(nodSursa+"-->"+k+" dist="+d[k]+" drum: ");
        drum(k);
      }
      else System.out.print(nodSursa+"-->"+k+" Nu exista drum! ");
      System.out.println();
    }

    out.close();
  }//main

  static void dijkstra(int nods)
  {
    int u,v,q,aux;

    d=new int [n+1];
    p=new int [n+1];
    hd=new int[n+1];
    hnod=new int[n+1];
    pozh=new int[n+1];

    for(u=1;u<=n;u++) { hnod[u]=pozh[u]=u; hd[u]=d[u]=oo; p[u]=0; }
    q=n;          // q = noduri nefinalizate = dimensiune heap
    d[nods]=0;
    hd[pozh[nods]]=0;

    urcInHeap(pozh[nods]);

    while(q>0)  // la fiecare pas adaug un varf in arbore
    {
      u=extragMin(q);

      if(u==-1) { System.out.println("graf neconex !!!"); break; }

      q--;
      for(v=1;v<=q;v++)         // noduri nefinalizate = in heap 1..q
        if(w[u][hnod[v]]<oo)    // cost finit ==> exista arc (u,v)
          relax(u,hnod[v]);     // relax si refac heap
    }
  }//dijkstra(...)

  static void relax(int u,int v)
  {
    if(d[u]+w[u][v]<d[v])
    {
```

```
      d[v]=d[u]+w[u][v];
      p[v]=u;
      hd[pozh[v]]=d[v];
      urcInHeap(pozh[v]);
    }
  }// relax(...)

  static int extragMin(int q) // in heap 1..q
  {
    // returnez valoarea minima (din varf!) si refac heap in jos
    int aux,fiu1,fiu2,fiu,tata,nod;

    aux=hd[1]; hd[1]=hd[q]; hd[q]=aux; // 1 <---> q
    aux=hnod[1]; hnod[1]=hnod[q]; hnod[q]=aux;
    pozh[hnod[q]]=q;
    pozh[hnod[1]]=1;

    tata=1;
    fiu1=2*tata;
    fiu2=2*tata+1;
    while(fiu1<=q-1)  //refac heap de sus in jos pana la q-1
    {
      fiu=fiu1;
      if(fiu2<=q-1)
        if(hd[fiu2]<hd[fiu1]) fiu=fiu2;

      if(hd[tata]<=hd[fiu]) break;

      pozh[hnod[fiu]]=tata;
      pozh[hnod[tata]]=fiu;
      aux=hd[fiu]; hd[fiu]=hd[tata]; hd[tata]=aux;
      aux=hnod[fiu]; hnod[fiu]=hnod[tata]; hnod[tata]=aux;

      tata=fiu;
      fiu1=2*tata;
      fiu2=2*tata+1;
    }

    return hnod[q]; // hnod[1] a ajuns deja (!) in hnod[q]
  } // extragMin(...)

  static void urcInHeap(int nodh)
  {
    int aux,fiu,tata,nod;
```

```
    nod=hnod[nodh];
    fiu=nodh;
    tata=fiu/2;
    while((tata>0)&&(hd[fiu]<hd[tata]))
    {
      pozh[hnod[tata]]=fiu;
      hnod[fiu]=hnod[tata];

      aux=hd[fiu]; hd[fiu]=hd[tata]; hd[tata]=aux;

      fiu=tata;
      tata=fiu/2;
    }

    pozh[nod]=fiu;
    hnod[fiu]=nod;
  } // urcInHeap(...)

  static void drum(int k) // s --> ... --> k
  {
    if(p[k]!=0) drum(p[k]);
    System.out.print(k+" ");
  }
}//class

/*
6 7           1-->1 dist=0 drum: 1
1 2 4         1-->2 dist=4 drum: 1 2
1 3 1         1-->3 dist=1 drum: 1 3
2 3 2         1-->4 dist=2 drum: 1 3 4
3 4 1         1-->5 Nu exista drum!
5 4 1         1-->6 dist=4 drum: 1 3 4 6
5 6 3
4 6 2
*/
```

14.3.14 Urgenţa - OJI2002 cls 11

Autorităţile dintr-o zonă de munte intenţionează să stabilească un plan de urgenţă pentru a reacţiona mai eficient la frecventele calamităţi naturale din zonă. În acest scop au identificat N puncte de interes strategic şi le-au numerotat distinct de la 1 la N. Punctele de interes strategic sunt conectate prin M căi de acces având

priorităţi în funcţie de importanţă. Între oricare două puncte de interes strategic există cel mult o cale de acces ce poate fi parcursă în ambele sensuri şi cel puţin un drum (format din una sau mai multe căi de acces) ce le conectează.

În cazul unei calamităţi unele căi de acces pot fi temporar întrerupte şi astfel între anumite puncte de interes nu mai există legătură. Ca urmare pot rezulta mai multe grupuri de puncte în aşa fel încât între oricare două puncte din acelaşi grup să existe măcar un drum şi între oricare două puncte din grupuri diferite să nu existe drum.

Autorităţile estimează gravitatea unei calamităţi ca fiind suma priorităţilor căilor de acces distruse de aceasta şi doresc să determine un scenariu de gravitate maximă, în care punctele de interes strategic să fie împărţite într-un număr de K grupuri.

Date de intrare

Fişierul de intrare URGENTA.IN are următorul format:

$N\ M\ K$

$i_1\ j_1\ p_1$ - între punctele i_1 şi j_1 există o cale de acces de prioritate p_1

$i_2\ j_2\ p_2$ - între punctele i_2 şi j_2 există o cale de acces de prioritate p_2

...

$i_M\ j_M\ p_M$ - între punctele i_M şi j_M există o cale de acces de prioritate p_M

Date de ieşire

Fişierul de ieşire URGENTA.OUT va avea următorul format:

$gravmax$ - gravitatea maximă

C - numărul de căi de acces întrerupte de calamitate

$k_1\ h_1$ - între punctele k_1 şi h_1 a fost întreruptă calea de acces

$k_2\ h_2$ - între punctele k_2 şi h_2 a fost întreruptă calea de acces

...

$k_C\ h_C$ - între punctele k_C şi h_C a fost întreruptă calea de acces

Restricţii şi precizări

$0 < N < 256$

$N - 2 < M < 32385$

$0 < K < N + 1$

Priorităţile căilor de acces sunt întregi strict pozitivi mai mici decât 256.

Un grup de puncte poate conţine între 1 şi N puncte inclusiv.

Dacă există mai multe soluţii, programul va determina una singură.

Exemplu

URGENTA.IN	URGENTA.OUT
7 11 4	27
1 2 1	8
1 3 2	1 3
1 7 3	1 7
2 4 3	2 4
3 4 2	3 4
3 5 1	3 7
3 6 1	4 5
3 7 5	5 6
4 5 5	6 7
5 6 4	
6 7 3	

Timp maxim de executare: 1 secundă / test

Codul sursă

```
import java.io.*; // arbore minim de acoperire: algoritmul lui Prim O(n^2)
class Urgenta     // sortare O(n^2) ... si una slaba merge!
{
  static final int oo=0x7fffffff;
  static int n,m,ncc,gravmax,costmax,nrm; // ncc = nr componente conexe
  static int[][] cost;
  static boolean[] esteInArbore;
  static int[] p;            // predecesor in arbore
  static int[] d;            // distante de la nod catre arbore
  static int[] a1;           // a1[k]=varful 1 al muchiei k din arbore
  static int[] a2;           // a2[k]=varful 2 al muchiei k din arbore
  static int[] ac;           // a1[k]=costul muchiei k din arbore

  public static void main(String[]args) throws IOException
  {
    int nodStart=3;          // nod start
    int i, j, k, costArbore=0,min,jmin=0,aux;

    StreamTokenizer st= new StreamTokenizer(
      new BufferedReader(new FileReader("urgenta.in")));
    PrintWriter out= new PrintWriter(
    new BufferedWriter(new FileWriter("urgenta.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); ncc=(int)st.nval;
```

```
cost=new int[n+1][n+1];
esteInArbore=new boolean [n+1];
p=new int[n+1];
d=new int[n+1];
a1=new int[n];
a2=new int[n];
ac=new int[n];

for(i=1;i<=n;i++) for(j=1;j<=n;j++) cost[i][j]=oo;
for(i=1;i<=n;i++) d[i]=oo;

costmax=0;
for(k=1;k<=m;k++)
{
  st.nextToken(); i=(int)st.nval;
  st.nextToken(); j=(int)st.nval;
  st.nextToken(); cost[i][j]=cost[j][i]=(int)st.nval;
  costmax+=cost[i][j];
}

// alg Prim
d[nodStart]=0;

for(k=1;k<=n;k++)    //  O(n)
{
  min=oo;
  for(j=1;j<=n;j++) // O(n)
  {
    if(esteInArbore[j]) continue;
    if(min>d[j]) { min=d[j]; jmin=j; }
  }//for j

  esteInArbore[jmin]=true;
  d[jmin]=0;
  costArbore+=min;

  for(j=1;j<=n;j++) // actualizez distantele nodurilor  //  O(n)
  {
    if(esteInArbore[j]) continue; // j este deja in arbore
    if(cost[jmin][j]<oo)          // am muchia (jmin,j)
      if(d[jmin]+cost[jmin][j]<d[j])
      {
        d[j]=d[jmin]+cost[jmin][j];
```

```
          p[j]=jmin;
        }
    }
  }//for k

  k=0;
  for(i=1;i<=n;i++)
    if(p[i]!=0)
    {
      //System.out.println(i+" "+p[i]+" --> "+cost[i][p[i]]);
      k++;
      a1[k]=i;
      a2[k]=p[i];
      ac[k]=cost[i][p[i]];
    }
  //System.out.println("cost="+costArbore);

  gravmax=costmax-costArbore; // deocamdata, ...

  //System.out.println("gravmax ="+gravmax);

  // trebuie sa adaug la gravmax primele ncc-1 costuri mari (sort!)
  // din arborele minim de acoperire

  // sortez descrescator ac (pastrand corect a1 si a2)
  // care are n-1 elemente
  for(k=1;k<=n-1;k++)        // de n-1 ori (bule)
  {
    for(i=1;i<=n-2;i++)
      if(ac[i]<ac[i+1])
      {
        aux=ac[i]; ac[i]=ac[i+1]; ac[i+1]=aux;
        aux=a1[i]; a1[i]=a1[i+1]; a1[i+1]=aux;
        aux=a2[i]; a2[i]=a2[i+1]; a2[i+1]=aux;
      }
  }

  // primele ncc-1 ...
  for(i=1;i<=ncc-1;i++) gravmax+=ac[i];

  // sterg muchiile ramase in arbore ...
  for(i=ncc;i<=n-1;i++)
  {
    cost[a1[i]][a2[i]]=cost[a2[i]][a1[i]]=oo; //sterg
```

```
    }

    out.println(gravmax);

    // determin numarul muchiilor ...
    nrm=0;
    for(i=1;i<=n-1;i++)
      for(j=i+1;j<=n;j++)
        if(cost[i][j] < oo)
          nrm++;
    out.println(nrm);

    // afisez muchiile ...
    for(i=1;i<=n-1;i++)
      for(j=i+1;j<=n;j++)
        if(cost[i][j] < oo)
          out.println(i+" "+j);

    out.close();
  }//main
}//class
```

14.3.15 Reactivi - OJI2004 cls 9

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 -10 10 - 2 5 20 50	2	4 2 5 5 7 10 20 30 40	3	5 -10 10 10 12 -20 10 7 10 7 8	2

Timp maxim de execuţie: 1 secundă/test

Indicaţii de rezolvare - descriere soluţie *

Soluţie prezentată de Mihai Stroe, GInfo nr. 14/4

Problema se poate rezolva prin metoda *greedy*.

O variantă mai explicită a enunţului este următoarea:

"Se consideră N intervale pe o axă. Să se aleagă un număr minim de puncte astfel încât fiecare interval să conţină cel puţin unul dintre punctele alese."

Facem o primă observaţie: pentru orice soluţie optimă există o soluţie cu acelaşi număr de puncte (frigidere), în care fiecare punct să fie sfârşitul unui interval. Aceasta se poate obţine mutând fiecare punct spre dreapta, până când ar ajunge la sfârşitul intervalului care se termină cel mai repede, dintre intervalele care îl conţin. Se observă că noua soluţie respectă restricţiile din enunţ.

În continuare ne vom concentra pe găsirea unei soluţii de acest tip.

Sortăm reactivii după sfârşitul intervalului. Pentru intervalul care se termină cel mai repede, alegem ultimul punct al său ca temperatură a unui frigider. Se observă că această alegere este cea mai bună, dintre toate alegerile unui punct în intervalul respectiv, în sensul că mulţimea intervalelor care conţin punctul este mai mare (conform relaţiei de incluziune), decât mulţimea corespunzătoare oricărei alte alegeri. Acest fapt este adevărat, deoarece mutarea punctului mai la stânga nu duce la selectarea unor noi intervale.

Intervalele care conţin punctul respectiv sunt eliminate (reactivii corespunzători pot fi plasaţi într-un frigider), iar procesul continuă cu intervalele rămase, în acelaşi

mod.

Analiza complexităţii

Notăm cu D numărul de temperaturi întregi din intervalul care conţine temperaturile din enunţ. Se observă că D este cel mult 201.

Citirea datelor de intrare are ordinul de complexitate $O(N)$.

Sortarea intervalelor după capătul din dreapta are ordinul de complexitate $O(N \cdot logN)$.

Urmează F paşi, unde F este numărul de frigidere selectate. Deoarece fiecare frigider este setat la o temperatură întreagă, $F \leq D$.

În cadrul unui pas, determinarea intervalului care se termină cel mai repede, pe baza vectorului sortat, are ordinul de complexitate $O(1)$. Eliminarea intervalelor care conţin un anumit punct (sfârşitul intervalului care se termină cel mai repede) are ordinul de complexitate $O(N)$.

Afişarea rezultatului are ordinul de complexitate $O(1)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N \cdot D + N \cdot logN)$; deoarece în general $D > logN$, considerăm ordinul de complexitate ca fiind $O(N \cdot D)$.

Codul sursă

```
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);
```

```
  }
}
```

14.3.16 Pal - ONI2005 cls 9

Autor: Silviu Gănceanu

Prinţul Algorel este în încurcătură din nou: a fost prins de Spânul cel Negru în încercarea sa de a o salva pe prinţesă şi acum este închis în Turnul cel Mare.

Algorel poate evada dacă găseşte combinaţia magică cu care poate deschide poarta turnului.

Prinţul ştie cum se formează această combinaţie magică: trebuie să utilizeze toate cifrele scrise pe uşa turnului pentru a obţine două numere palindroame, astfel încât suma lor să fie minimă, iar această sumă este combinaţia magică ce va deschide uşa.

Primul număr palindrom trebuie să aibă cel puţin L cifre, iar cel de-al doilea poate avea orice lungime diferită de 0. Numerele palindroame formate nu pot începe cu cifra 0. Acum interveniţi dumneavoastră în poveste, fiind prietenul său cel mai priceput în algoritmi.

Prin noul super-telefon al său, prinţul transmite numărul de apariţii a fiecărei cifre de pe uşa turnului precum şi lungimea minimă L a primului număr, iar dumneavoastră trebuie să-i trimiteţi cât mai repede numerele cu care poate obţine combinaţia magică.

Cerinţă

Având datele necesare, aflaţi două numere palindroame cu care se poate obţine combinaţia magică.

Date de intrare

Prima linie a fişierului **pal.in** conţine un număr întreg L reprezentând lungimea minimă a primului număr. Urmează 10 linii: pe linia $i + 2$ se va afla un număr întreg reprezentând numărul de apariţii ale cifrei i, pentru i cu valori de la 0 la 9.

Date de ieşire

Prima linie a fişierului de ieşire **pal.out** conţine primul număr palidrom, iar cea de-a doua linie conţine cel de-al doilea număr palindrom. Dacă există mai multe soluţii se va scrie doar una dintre ele.

Restricţii şi precizări

• În total vor fi cel mult 100 de cifre

• $1 \leq L < 100$ şi L va fi mai mic decât numărul total de cifre

• Pentru datele de test va exista întotdeauna soluţie: se vor putea forma din cifrele scrise pe uşa turnului două numere care încep cu o cifră diferită de 0, iar primul număr să aibă cel puţin L cifre

• Un număr este palindrom dacă el coincide cu răsturnatul său. De exemplu 12321 şi 7007 sunt numere palindroame, în timp ce 109 şi 35672 nu sunt.

• Pentru 30% dintre teste, numărul total de cifre va fi cel mult 7; pentru alte 40% din teste numărul total de cifre va fi cel mult 18, iar pentru restul de 30% din teste numărul total de cifre va fi mai mare sau egal cu 30.

• Fiecare linie din fişierul de intrare şi din fişierul de ieşire se termină cu marcaj de sfârşit de linie.

Exemplu

pal.in	**pal.out**	Explicaţie
5	10001	Pentru acest exemplu avem $L = 5$,
3	222	3 cifre de 0, 2 cifre de 1şi 3 cifre de 2.
2		Cifrele de la 3 la 9 lipsesc
3		de pe uşa turnului.
0		
0		Cele două palindroame cu care
0		se generează combinaţia magică
0		sunt 10001 şi 222.
0		Combinaţia magică va fi suma acestora
0		şi anume 10223 (care este suma minimă
0		pe care o putem obţine).

Timp maxim de execuţie/test: 1 sec sub Windows şi 1 sec sub Linux

Indicaţii de rezolvare - descriere soluţie

Soluţia oficială, Silviu Gănceanu

Problema se rezolvă utilizând tehnica greedy. Notăm numărul total de cifre cu NC. Se observă că, pentru ca suma celor două numere palindroame să fie minimă, trebuie ca lungimea maximă a celor două numere să fie cât mai mică. Aşadar, pentru început, se stabileşte lungimea exactă a primului număr (care va fi cel mai lung), în funcţie de L în modul următor:

1. dacă $L < NC/2$, atunci lungimea primului palindrom va fi aleasă astfel încât cele două numere să fie cât mai apropiate ca lungime

2. dacă $L >= NC/2$ apar cazuri particulare şi se stabileşte dacă lungimea primului palindrom creşte sau nu cu o unitate.

Având lungimile numerelor stabilite putem merge mai departe utilzând strategia greedy, observând că cifrele mai mici trebuie să ocupe poziţii cât mai semnificative. Pentru a realiza acest lucru se vor completa în paralel cele două numere cu cifrele parcurse în ordine crescătoare stabilind în care din cele două numere se vor poziţiona. De asemenea, trebuie avut în vedere ca niciunul din cele două numere să nu înceapă cu cifra 0.

Datele de test au fost construite astfel încât şi soluţii neoptime să obţină puncte, gradat, în funcţie de optimizările efectuate asupra implementării.

Codul sursă

```
import java.io.*;  // la inceput cifre mici in p1
class Pal          // apoi in ambele in paralel!
{
  static int L;
  static int NC=0;
  static int nc1,nc2;
  static int ncfi=0;           // nr cifre cu frecventa impara
  static int[] fc=new int[10]; // frecventa cifrelor
  static int[] p1;             // palindromul 1
  static int[] p2;             // palindromul 2

  public static void main(String []args) throws IOException
  {
    int i;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("pal.in")));
    PrintWriter out =  new PrintWriter (
                       new BufferedWriter( new FileWriter("pal.out")));

    st.nextToken();L=(int)st.nval;
    for(i=0;i<=9;i++) { st.nextToken(); fc[i]=(int)st.nval;}

    for(i=0;i<=9;i++) NC+=fc[i];           // nr total cifre
    for(i=0;i<=9;i++) ncfi=ncfi+(fc[i]%2); // nr cifre cu frecventa impara

    nc1=L;
    nc2=NC-nc1;
    while(nc1<nc2) {nc1++; nc2--;}
    if((ncfi==2)&&(nc1%2==0)) {nc1++; nc2--;}

    p1=new int[nc1];
    p2=new int[nc2];
    switch(ncfi)
    {
      case 0: impare0(); break;
      case 1: impare1(); break;
      case 2: impare2(); break;
      default: System.out.println("Date de intrare eronate!");
    }
    corectez(p1);
    corectez(p2);
```

```
  for(i=0;i<p1.length;i++) out.print(p1[i]);
  out.println();
  for(i=0;i<p2.length;i++) out.print(p2[i]);
  out.println();
  int[] p12=suma(p1,p2);// pentru ca in teste rezultat = suma!
  for(i=p12.length-1;i>=0;i--) out.print(p12[i]);
  out.println();
  out.close();
}//main

static void impare0() // cea mai mare cifra la mijloc
{
  int i,k1,k2;
  int imp1=nc1/2, imp2=nc2/2;

  if(nc1%2==1) // numai daca nc1 si nc2 sunt impare !
    for(i=9;i>=0;i--)
      if(fc[i]>0)
      {
        p1[imp1]=i; fc[i]--;
        p2[imp2]=i; fc[i]--;
        break;
      }
  k1=0;
  k2=0;
  while(k1<nc1-nc2) {incarcPozitia(k1,p1); k1++;}// incarc numai p1
  while((k1<imp1)||(k2<imp2))
  {
    if(k1<imp1) {incarcPozitia(k1,p1); k1++;}
    if(k2<imp2) {incarcPozitia(k2,p2); k2++;}
  }
}

static void impare1()
{
  int i,k1,k2;
  int imp1=nc1/2, imp2=nc2/2;

  for(i=0;i<=9;i++)
    if(fc[i]%2==1) { p1[imp1]=i; fc[i]--; break;}
  for(i=0;i<=9;i++)
    if(fc[i]%2==1) { p2[imp2]=i; fc[i]--; break;}
```

```
    k1=0;
    k2=0;
    while(k1<nc1-nc2) {incarcPozitia(k1,p1); k1++;} // incarc numai p1
    while((k1<imp1)||(k2<imp2))
    {
      if(k1<imp1) {incarcPozitia(k1,p1); k1++;}
      if(k2<imp2) {incarcPozitia(k2,p2); k2++;}
    }
  }

  static void impare2()
  {
    int i,k1,k2;
    int imp1=nc1/2, imp2=nc2/2;

    for(i=0;i<=9;i++)
      if(fc[i]%2==1) { p1[imp1]=i; fc[i]--; break;}
    for(i=0;i<=9;i++)
      if(fc[i]%2==1) { p2[imp2]=i; fc[i]--; break;}

    k1=0;
    k2=0;
    while(k1<nc1-nc2) {incarcPozitia(k1,p1); k1++;}// incarc numai p1
    while((k1<imp1)||(k2<imp2))
    {
      if(k1<imp1) { incarcPozitia(k1,p1); k1++; }
      if(k2<imp2) { incarcPozitia(k2,p2); k2++; }
    }
  }

  static void corectez(int[] x)
  {
    int pozdif0,val, n=x.length;
    pozdif0=0;
    while(x[pozdif0]==0) pozdif0++;
    if(pozdif0>0)
    {
      val=x[pozdif0];
      x[0]=x[n-1]=val;
      x[pozdif0]=x[n-pozdif0-1]=0;
    }
  }

  static void incarcPozitia(int k, int[] x)
```

```
  {
    int i;
    int n=x.length;
    for(i=0;i<=9;i++)
      if(fc[i]>0)
      {
        x[k]=i; fc[i]--;
        x[n-k-1]=i; fc[i]--;
        break;
      }
  }

  static int[] suma(int[] x, int[] y)
  {
    int[] z=new int[(x.length>y.length) ? (x.length+1) : (y.length+1)];
    int k,t=0;
    for(k=0;k<=z.length-2;k++)
    {
      z[k]=t;
      if(k<x.length) z[k]+=x[k];
      if(k<y.length) z[k]+=y[k];
      t=z[k]/10;
      z[k]=z[k]%10;
    }
    z[z.length-1]=t;
    if(z[z.length-1]!=0) return z;
    else
    {
      int[] zz=new int[z.length-1];
      for(k=0;k<zz.length;k++) zz[k]=z[k];
      return zz;
    }
  }
}//class
```

14.3.17 Şanţ - ONI2006 cls 9

Cei n deţinuţi ai unei închisori, numerotaţi de la 1 la n, trebuie să sape un şanţ dispus în linie dreaptă între două puncte A şi B, situate la distanţa de 250 km unul de celălalt, pe care există 251 borne kilometrice numerotate de la 0 la 250. Fiecare deţinut are însă o pretenţie, "e doar democraţie, nu?": el doreşte să sape doar undeva în zona dintre borna x şi borna y. Pe lângă aceste pretenţii închisoarea se confruntă şi cu o altă problemă: nu are suficienţi paznici angajaţi.

Cerinţă

Cunoscându-se numărul n de deţinuţi şi pretenţiile lor, să se determine locul (locurile) unde vor fi puşi deţinuţii să sape într-o zi de muncă, respectându-se pretenţiile lor, astfel încât numărul de paznici necesari pentru a păzi cei n deţinuţi să fie minim. Intervalul în care poate păzi un paznic nu poate conţine două sau mai multe zone disjuncte dintre cele exprimate de deţinuţi în preferinţele lor.

Date de intrare

Fişierul de intrare **sant.in** are pe prima linie numărul n de deţinuţi. Pe fiecare dintre următoarele n linii există câte două numere naturale, a_i b_i, separate printr-un spaţiu ($a_i \leq b_i$), care reprezintă pretenţia deţinutului. Mai exact pe linia $i+1$ ($1 \leq i \leq n$) este descrisă pretenţia deţinutului cu numărul de ordine i.

Date de ieşire

Fişierul de ieşire **sant.out** va conţine pe prima linie numărul natural k reprezentând numărul minim de paznici necesari pentru paza celor n deţinuţi scoşi la lucru. Pe următoarele $2k$ linii vor fi descrise locurile unde vor fi puşi să sape deţinuţii, astfel: fiecare pereche de linii $(2j, 2j+1)$ va conţine pe linia $2j$ trei numere naturale p x_j y_j, separate prin câte un spaţiu reprezentând numărul de ordine al paznicului şi bornele kilometrice x_j c si y_j unde va păzi paznicul p, iar pe linia $2j+1$ vor fi scrise numerele de ordine ale deţinuţilor care sapă în această zonă, separate prin câte un spaţiu, ordonate crescător.

Restricţii şi precizări

- $1 \leq n \leq 10000$
- $0 \leq a_i \leq b_i \leq 250$, pentru orice i, $1 \leq i \leq n$
- $0 \leq x_j \leq y_j \leq 250$, pentru orice j, $1 \leq j \leq k$
- un deţinut poate să sape şi într-un singur punct ("în dreptul bornei kilometrice numerotată cu x")
- în cazul în care există mai multe soluţii se va afişa una singură
- numerele de ordine ale paznicilor vor fi scrise în fişier în ordine crescătoare
- numerotarea paznicilor începe de la 1

Exemplu

.in	.out	Explicaţie
3 0 20 8 13 30 60	2 1 8 13 1 2 2 30 60 3	sunt necesari 2 paznici: paznicul 1 va păzi între borna 8 şi borna 13, iar deţinuţii păziţi sunt 1 şi 2; paznicul 2 va păzi între borna 30 şi borna 60, iar deţinutul păzit este 3
4 10 203 2 53 30 403 5 7 33	3 1 10 20 1 2 5 5 2 4 3 30 40 3	sunt necesari 3 paznici: paznicul 1 va păzi între borna 10 şi borna 20, iar deţinutul păzit este 1; paznicul 2 va păzi la borna 5, iar deţinuţii păziţi sunt 2 şi 4; paznicul 3 va păzi între borna 30 şi borna 40, iar deţinutul păzit este 3
5 10 30 30 32 0 30 27 30 27 28	2 1 30 30 1 2 3 4 2 27 28 5	sunt necesari 2 paznici: paznicul 1 va păzi la borna 30, iar deţinuţii păziţi sunt 1, 2, 3 şi 4; paznicul 2 va păzi între borna 27 şi borna 28, iar deţinutul păzit este 5 *!Soluţia nu este unică!*

Timp maxim de execuţie/test: 1 secundă (Windows), 0.5 secunde (Linux)

Indicaţii de rezolvare - descriere soluţie

Soluţia comisiei

Problema cere, de fapt, determinarea numărului minim de intersecţii între segmentele determinate de kilometrul minim şi maxim între care sapă un deţinut.

Pentru a le determina procedez astfel:

- ordonez intervalele crescător după kilometrul minim şi descrescător după kilometrul maxim
- pun primul deţinut (deci cel cu intervalul de săpare cel mai mare) în grija primului paznic
- pentru toate celelalte
  - caut un paznic care mai păzeşte deţinuţi şi care poate păzi şi acest deţinut (adică intersecţia segmentelor să fie nevidă)
  - dacă găsesc
  - ajustez intervalul de săpare ca să poată săpa şi acest deţinut
  - altfel (dacă nu găsesc)
  - mai am nevoie de un paznic şi îl dau în grija acestuia

Datorită faptului că intervalele sunt sortate, crescător după capătul inferior, în momentul când un interval nu se mai intersectează cu cel deja determinat, este sigur că nici un alt interval ce îl urmează nu se mai intersectează, lucru ce asigură determinarea numărului minim de paznici.

Codul sursă

```
import java.io.*;
class Sant
{
  static int n;
  static int[] x1=new int[10001];
  static int[] x2=new int[10001];
  static int[]  o=new int[10001];

  static void qsort(int li,int ls)
  {
    int val,aux,i,j;

    i=li;
    j=ls;
    val=x2[(i+j)/2];

    while(i<j)
    {
      while(x2[i]<val) i++;
      while(x2[j]>val) j--;
      if(i<=j)
      {
        aux=x1[i]; x1[i]=x1[j]; x1[j]=aux;
        aux=x2[i]; x2[i]=x2[j]; x2[j]=aux;
        aux=o[i];  o[i]=o[j];    o[j]=aux;
        i++;
        j--;
      }
    }// while

    if(li<j) qsort(li,j);
    if(i<ls) qsort(i,ls);
  }// qsort(...)

  public static void main(String[] args) throws IOException
```

```
  {
    int k,np,xp;
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("sant.in")));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("sant.out")));

    st.nextToken(); n=(int)st.nval;
    for(k=1;k<=n;k++)
    {
      st.nextToken(); x1[k]=(int)st.nval;
      st.nextToken(); x2[k]=(int)st.nval;
      o[k]=k;
    }

    qsort(1,n);

    np=1;
    xp=x2[1];
    for(k=2;k<=n;k++) if(x1[k]>xp) { np++; xp=x2[k]; }
    out.println(np);

    // inca o data pentru ...!
    np=1;
    xp=x2[1];
    out.println(np+" "+xp+" "+xp);
    out.print(o[1]+" ");
    for(k=2;k<=n;k++)
    {
      if(x1[k]>xp)
      {
        out.println();
        np++;
        xp=x2[k];
        out.println(np+" "+xp+" "+xp);
        out.print(o[k]+" ");
      }
      else out.print(o[k]+" ");
    }// for k
    out.close();
  }// main(...)
}// class
```

14.3.18 Cezar - OJI2007 cls 11

În Roma antică există n aşezări senatoriale distincte, câte una pentru fiecare dintre cei n senatori ai Republicii. Aşezările senatoriale sunt numerotate de la 1 la n, între oricare două aşezări existând legături directe sau indirecte. O legătură este directă dacă ea nu mai trece prin alte aşezări senatoriale intermediare. Edilii au pavat unele dintre legăturile directe dintre două aşezări (numind o astfel de legătură pavată "stradă"), astfel încât între oricare două aşezări senatoriale să existe o singură succesiune de străzi prin care se poate ajunge de la o aşezare senatorială la cealaltă.

Toţi senatorii trebuie să participe la şedinţele Senatului. În acest scop, ei se deplasează cu lectica. Orice senator care se deplasează pe o stradă plăteşte 1 ban pentru că a fost transportat cu lectica pe acea stradă.

La alegerea sa ca prim consul, Cezar a promis că va dota Roma cu o lectică gratuită care să circule pe un număr de k străzi ale Romei astfel încât orice senator care va circula pe străzile respective, să poată folosi lectica gratuită fără a plăti. Străzile pe care se deplasează lectica gratuită trebuie să fie legate între ele (zborul, metroul sau teleportarea nefiind posibile la acea vreme).

În plus, Cezar a promis să stabilească sediul sălii de şedinţe a Senatului într-una dintre aşezările senatoriale aflate pe traseul lecticii gratuite. Problema este de a alege cele k străzi şi amplasarea sediului sălii de şedinţe a Senatului astfel încât, prin folosirea transportului gratuit, senatorii, în drumul lor spre sala de şedinţe, să facă economii cât mai însemnate. În calculul costului total de transport, pentru toţi senatorii, Cezar a considerat că fiecare senator va călători exact o dată de la aşezarea sa până la sala de şedinţe a Senatului.

Cerinţă

Scrieţi un program care determină costul minim care se poate obţine prin alegerea adecvată a celor k străzi pe care va circula lectica gratuită şi a locului de amplasare a sălii de şedinţă a Senatului.

Date de intrare

Fişierul **cezar.in** conţine

• pe prima linie două valori n k separate printr-un spaţiu reprezentând numărul total de senatori şi numărul de strazi pe care circulă lectica gratuită

• pe următorele $n-1$ linii se află câte două valori i j separate printr-un spaţiu, reprezentând numerele de ordine a două aşezări senatoriale între care există stradă.

Date de ieşire

Pe prima linie a fişierului **cezar.out** se va scrie costul total minim al transportării tuturor senatorilor pentru o alegere optimă a celor k străzi pe care va circula lectica gratuită şi a locului unde va fi amplasată sala de şedinţe a Senatului.

Restricţii şi precizări

• $1 < n \leq 10000$, $0 < k < n$

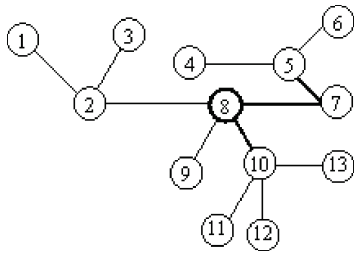
• $1 \leq i, j \leq n$, $i \neq j$

• Oricare două perechi de valori de pe liniile $2, 3, ..., n$ din fişierul de intrare reprezintă două străzi distincte.

• Perechile din fişierul de intrare sunt date astfel încât respectă condiţiile din problemă.

• Pentru 25% din teste $n \leq 30$, pentru 25% din teste $30 < n \leq 1000$, pentru 25% din teste $1000 < n \leq 3000$, pentru 10% din teste $3000 < n \leq 5000$, pentru 10% din teste $5000 < n \leq 10000$.

Exemplu



cezar.in	**cezar.out**	**Explicaţie**
13 3 1 2 2 3 2 8 7 8 7 5 5 4 5 6 8 9 8 10 10 11 10 12 10 13	11	Costul minim se obţine, de exemplu, pentru alegerea celor 3 străzi între aşezările 5-7, 7-8, 8-10 şi a sălii de şedinţe a Senatului în aşezarea 8 (după cum este evidenţiat în desen). Există şi alte alegeri pentru care se obţine soluţia 11.

Timp maxim de execuţie/test: 0.5 secunde

Indicaţii de rezolvare - descriere soluţie *

O implementare posibilă utilizează metoda GREEDY, $O(n * (n - k))$ sau $O((n - k) * log(n))$.

Se elimină succesiv, dintre frunzele existente la un moment dat, frunza de cost minim. Toate nodurile au costul iniţial 1. La eliminarea unei frunze, se incrementează cu 1 costul tatălui acesteia. Validitatea metodei rezultă din observaţia că, la eliminarea unei frunze oarecare, tatăl acesteia poate deveni frunză la rândul lui, dar cu un cost strict mai mare decât al frunzei eliminate.

Se poate reţine arborele cu ajutorul listelor de adiacenţă (liniare sau organizate ca arbori de căutare), iar frunzele se pot memora într-un minheap de costuri, structură care se actualizează în timp logaritmic.

Codul sursă

Varianta 1:

```
import java.io.*;
class cezar
{
  static StreamTokenizer st;
  static PrintWriter out;

  static int n, ns;

  static int[] v1 = new int[10001];
  static int[] v2 = new int[10001];
  static int[]  g = new int[10001];     // gradele
  static int[] ca = new int[10001];     // ca[k]=costul acumulat in nodul k !!!
  static int[] nde = new int[10001];    // nde[k]=nr "descendenti" eliminati pana in k

  static void citire() throws IOException
  {
    int i,j,k;

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); ns=(int)st.nval;

    for(i=1;i<=n;i++) g[i]=ca[i]=nde[i]=0;  // curatenie ...

    for(k=1;k<=n-1;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;

      v1[k]=i;
      v2[k]=j;
```

```
    g[i]++; g[j]++;
  }
}// citire(...)

static int tata(int i)  // mai bine cu lista de adiacenta ...
{
  int k,t;
  t=-1;                 // initializarea aiurea ...
  for(k=1;k<=n-1;k++)  // n-1 muchii
  {
    if( (v1[k]==i) && (g[v2[k]] > 0) ) {t=v2[k]; break;}
    else
    if( (v2[k]==i) && (g[v1[k]] > 0) ) {t=v1[k]; break;}
  }
  return t;
}

static void eliminm()    // frunze(g=1) cu cost=min
{
  int i, imin, timin;
  int min;

  min=Integer.MAX_VALUE;
  imin=-1;              // initializare aiurea ...
  for(i=1;i<=n;i++)     // mai bine cu heapMIN ...
    if(g[i]==1)         // i=frunza
    if(nde[i]+1<min)    // ... aici este testul CORECT ... !!!
    {
      min=nde[i]+1;
      imin=i;
    }

  timin=tata(imin);
  g[imin]--; g[timin]--;
  ca[timin]=ca[timin]+ca[imin]+nde[imin]+1;
  nde[timin]=nde[timin]+nde[imin]+1; // nr descendenti eliminati
}// elimin()

public static void main(String[] args) throws IOException
{
  int k,senat=0,cost=0;
  st=new StreamTokenizer(new BufferedReader(new FileReader("cezar20.in")));
  out=new PrintWriter(new BufferedWriter(new FileWriter("cezar.out")));
```

```
    citire();

    for(k=1;k<=n-1-ns;k++) eliminm();

    for(k=1;k<=n;k++)
      if(g[k]>0)
      {
        cost+=ca[k];
        senat=k;
      }

    System.out.println("\n"+cost+" "+senat);
    out.println(cost+" "+senat);
    out.close();
    //afisv(g,1,n);
  }// main
}// class
```

Varianta 2:

```
import java.io.*;
class cezar               // cost initial = 1 pentru toate nodurile ...
{
  static StreamTokenizer st;
  static PrintWriter out;

  static int n, ns,s=0;

  static int[] v1 = new int[10001];
  static int[] v2 = new int[10001];
  static int[]  g = new int[10001];         // gradele
  static int[] ca = new int[10001];         // ca[k]=costul acumulat in nodul k !!!

  static void afisv(int[] v,int i1, int i2)
  {
    int i;
    for(i=i1;i<=i2;i++)
    {
      System.out.print(v[i]+" ");
      if(i%50==0) System.out.println();
    }
    System.out.println();
  }
```

```
static void citire() throws IOException
{
  int i,j,k;

  st.nextToken(); n=(int)st.nval;
  st.nextToken(); ns=(int)st.nval;

  for(i=1;i<=n;i++)              // curatenie ...
  {
    g[i]=0;
    ca[i]=1;                    // cost initial in nodul i
  }

  for(k=1;k<=n-1;k++)
  {
    st.nextToken(); i=(int)st.nval;
    st.nextToken(); j=(int)st.nval;

    v1[k]=i;
    v2[k]=j;
    g[i]++; g[j]++;
  }

  //afisv(v1,1,n-1);
  //afisv(v2,1,n-1);
  //afisv(g,1,n);
}// citire(...)

static int tata(int i)   // mai bine cu liste de adiacenta ...
{
  int k,t;
  t=-1;                  // initializarea aiurea ...
  for(k=1;k<=n-1;k++)    // este mai bine cu lista de adiacenta ?
  {
    if( (v1[k]==i) && (g[v2[k]]>0)) {t=v2[k]; break;}
    else
    if( (v2[k]==i) && (g[v1[k]]>)) {t=v1[k]; break;}
  }
  return t;
}

static void eliminm()     // frunze(g=1) cu cost=min
{
```

```
    int i, imin, timin;
    int min;

    min=Integer.MAX_VALUE;
    imin=-1;                // initializare aiurea ...
    for(i=1;i<=n;i++)       // mai bine cu heapMIN ...
      if(g[i]==1)           // i=frunza
      if(ca[i]<min)         // cost acumulat
      {
        min=ca[i];
        imin=i;
      }

    timin=tata(imin);

    g[imin]--; g[timin]--;

    ca[timin]=ca[timin]+min;
    s+=min;

    //System.out.println(" Elimin nodul "+imin+
    //                   " timin = "+timin+"  ca = "+ca[timin]);
  }// elimin()

  public static void main(String[] args) throws IOException
  {
    int k,senat=0;
    st=new StreamTokenizer(new BufferedReader(new FileReader("cezar20.in")));
    out=new PrintWriter(new BufferedWriter(new FileWriter("cezar.out")));

    citire();

    for(k=1;k<=n-1-ns;k++)
    {
      eliminm();
    }

    //afisv(c,1,n);

    for(k=1;k<=n;k++)
      if(g[k]>0)
      {
        senat=k;
        break;
```

```
      }

    System.out.println("\n"+s+" "+senat);

    out.println(s+" "+senat);
    out.close();
  }// main
}// class
```

Capitolul 15

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.

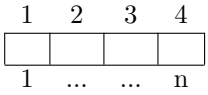
15.1 Generarea produsului cartezian

Pentru începători, nu metoda backtracking în sine este dificil de înţeles ci modalitatea de generare a produsului cartezian.

15.1.1 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 $S_1 \times S_2 \times ... \times S_n$

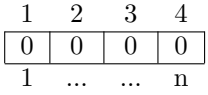
(de exemplu, pentru $n = 4$). Folosim un vector cu n elemente $a = (a_1, a_2, ..., a_n)$ şi vom atribui fiecărei componente a_i valori cuprinse între min şi max.



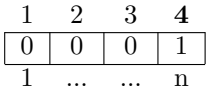
Ne trebuie un *marcaj* pentru a preciza faptul că o anumită componentă este *goală* (nu are plasată în ea o valoare validă). În acest scop folosim variabila `gol` care poate fi iniţializată cu orice valoare întreagă care nu este în intervalul $[min, max]$. Este totuşi utilă iniţializarea `gol=min-1;`.

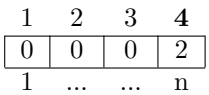
Cum începem generarea produsului cartezian?

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.

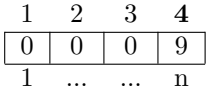


Trebuie să construim următoarele elemente ale produsului cartezian. Este utilă, în acest moment, imaginea kilometrajelor de maşini sau a contoarelor de energie electică, de apă, de gaze, etc. Următoarea configuraţie a acestora este

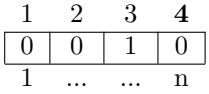




Folosim o variabilă k pentru a urmări poziţia din vector pe care se schimbă valorile. La început $k = n$ şi, pe această poziţie k, valorile se schimbă crescând de la min către max. Se ajunge astfel la configuraţia ($k = 4 = n$ aici!)

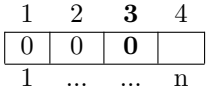


Ce se întâmplă mai departe? Apare configuraţia



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.

0	1	2	**3**	4	5
	0	0	**1**		
0	1	...	...	n	n+1

După plasarea unei valori pe poziţia k, se face pasul spre dreapta ($k = k+1$).

0	1	2	3	**4**	5
	0	0	1		
0	1	...	...	n	n+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).

0	1	2	3	**4**	5
	0	0	1	**0**	
0	1	...	...	n	n+1

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

0	1	2	3	**4**	5
	0	0	9	**9**	
0	1	...	...	n	n+1

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 $k = k - 1$

0	1	2	**3**	4	5
	0	0	**9**		
0	1	...	...	n	n+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 $k = k - 1$

0	1	**2**	3	4	5
	0	**0**			
0	1	...	...	n	n+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 $k = k + 1$

0	1	2	**3**	4	5
	0	1			
0	1	...	...	n	n+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");
  }
}
```

15.1.2 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
```

15.2 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 $A_1 \times A_2 \times ... \times A_n$;

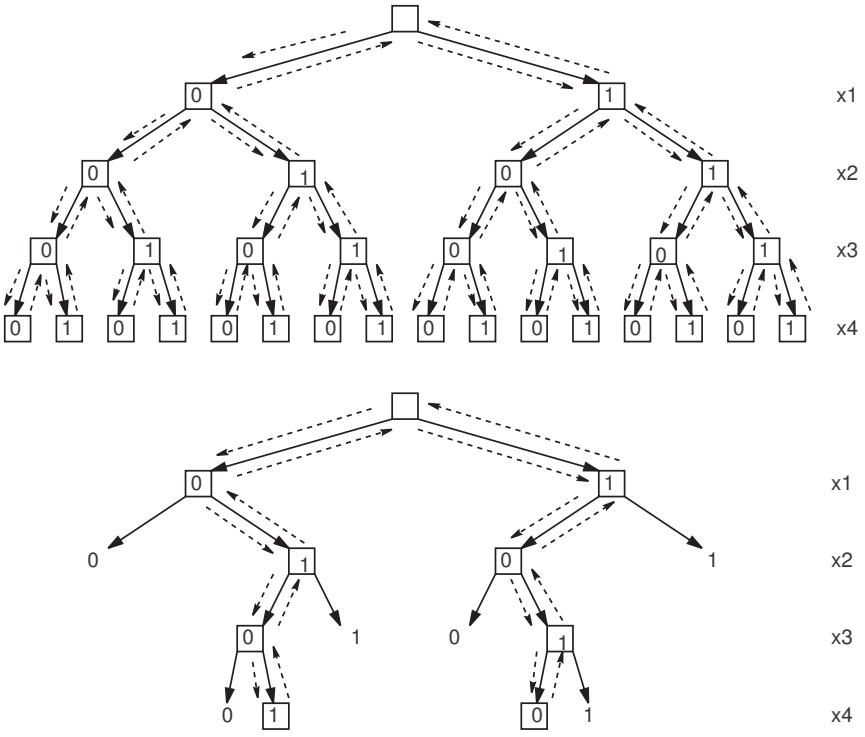
• la completarea componentei k se verifică dacă soluţia parţială $(x_1, x_2, ..., x_k)$ verifică condiţiile induse de restricţiile problemei (acestea sunt numite *condiţii de continuare*);

• 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).

15.2.1 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.

15.2.2 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.

15.3 Probleme rezolvate

15.3.1 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();
  }
}
```

15.3.2 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
```

15.3.3 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();
  }
}
```

15.3.4 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
```

15.3.5 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();
  }
}
```

15.3.6 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
```

15.3.7 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
```

15.3.8 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
```

15.3.9 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
```

15.3.10 Algoritmul DFS de parcurgere a grafurilor

Algoritmul DFS de parcurgerea in adâncime a grafurilor neorientate foloseste tehnica backtrackink.

```
import java.io.*;  // arborele DFS (parcurgere in adancime)
class DFS          // momentele de descoperire si finalizare a nodurilor
{                  // drum intre doua varfuri
  static final int WHITE=0, GRAY=1, BLACK=2;
  static int n,m,t;
  static int[] d,f,p,color;  // descoperit,finalizat,predecesor,culoare
  static int[][] a;

  public static void main(String[] args) throws IOException
  {
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("dfs.in")));

    int i,j,k,nods,nodd; // nods=nod_start_DFS, nod_destinatie (drum!)

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); nods=(int)st.nval;
    st.nextToken(); nodd=(int)st.nval;

    a=new int[n+1][n+1];
    d=new int[n+1];
    f=new int[n+1];
    p=new int[n+1];
    color=new int[n+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=1;
      a[j][i]=1;
    }

    for(i=1;i<=n;i++) // oricum erau initializati implicit, dar ... !!!
    {
      color[i]=WHITE;
      p[i]=-1;
    }
```

```
    t=0;
    dfs(nods);

    System.out.print("drum : "); drum(nodd); System.out.println();
    System.out.print("Descoperit :\t"); afisv(d);
    System.out.print("Finalizat  :\t"); afisv(f);
  }//main

  static void dfs(int u)
  {
    int v;
    color[u]=GRAY;
    d[u]=++t;
    for(v=1;v<=n;v++)       // listele de adiacenta ... !!!
    if(a[u][v]==1)          // v in Ad(u) !!!
      if(color[v]==WHITE)
      {
        p[v]=u;
        dfs(v);
      }
    color[u]=BLACK;
    f[u]=++t;
  }//dfs

  static void drum(int u)   // nod_sursa ---> nod_destinatie
  {
    if(p[u]!=-1) drum(p[u]);
    System.out.print(u+" ");
  }// drum(...)

  static void afisv(int[] x)
  {
    int i;
    for(i=1;i<=n;i++) System.out.print(x[i]+"\t");
    System.out.println();
  }
}//class

/*
6 7 3 4       drum : 3 1 4
1 4           Descoperit :    2   5    1    3    4    8
4 6           Finalizat  :   11   6   12   10    7    9
6 1
```

```
5 3
2 5
1 3
4 5
*/
```

15.3.11 Determinarea componentelor conexe

```
import java.io.*; // determinarea componentelor conexe
class CompConexe
{
  static int n,m,ncc;
  static int [] cc;
  static int[][] a;

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("compConexe.in")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    a=new int[n+1][n+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=a[j][i]=1;
    }

    cc=new int[n+1];

    ncc=0;
    for(i=1;i<=n;i++)
      if(cc[i]==0)
      {
        ncc++;
        conex(i);
      }

    for(i=1;i<=ncc;i++)
```

```
    {
      System.out.print(i+" : ");
      for(j=1;j<=n;j++)
        if(cc[j]==i)
          System.out.print(j+" ");
      System.out.println();
    }
  }//main

  static void conex(int u)
  {
    cc[u]=ncc;
    for(int v=1;v<=n;v++)
      if((a[u][v]==1)&&(cc[v]==0))
        conex(v);
  }//conex
}//class

/*
9 7       1 : 1 2 3
1 2       2 : 4 5
2 3       3 : 6 7 8 9
3 1
4 5
6 7
7 8
8 9
*/
```

15.3.12 Determinarea componentelor tare conexe

```
// determinarea componentelor tare conexe (in graf orientat!)
// Algoritm:  1. dfs(G) pentru calculul f[u]
//            2. dfs(G_transpus) in ordinea descrescatoare a valorilor f[u]
// OBS: G_transpus are arcele din G "intoarse ca sens"
//      Lista este chiar o sortare topologica !!!

import java.io.*;
class CompTareConexe
{
  static final int WHITE=0, GRAY=1, BLACK=2;
  static int n,m,t=0,nctc,pozLista;
  static int [] ctc,f,color,lista;
  static int[][] a;   // matricea grafului
```

```
  static int[][] at;  // matricea grafului transpus (se poate folosi numai a !)

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("compTareConexe.in")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    a=new int[n+1][n+1];  at=new int[n+1][n+1];   ctc=new int[n+1];
    f=new int[n+1];       lista=new int[n+1];     color=new int[n+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=1;
      at[j][i]=1; // transpusa
    }

    for(i=1;i<=n;i++) color[i]=WHITE;

    pozLista=n;
    for(i=1;i<=n;i++) if(color[i]==WHITE) dfsa(i);

    nctc=0;
    for(i=1;i<=n;i++) color[i]=WHITE;
    for(i=1;i<=n;i++)
      if(color[lista[i]]==WHITE) { nctc++; dfsat(lista[i]); }

    for(i=1;i<=nctc;i++)
    {
      System.out.print(i+" : ");
      for(j=1;j<=n;j++) if(ctc[j]==i) System.out.print(j+" ");
      System.out.println();
    }
  }//main

  static void dfsa(int u)
  {
    int v;
    color[u]=GRAY;
```

```
    for(v=1;v<=n;v++) if((a[u][v]==1)&&(color[v]==WHITE)) dfsa(v);
    color[u]=BLACK;   f[u]=++t;  lista[pozLista--]=u;
  }

  static void dfsat(int u) // se poate folosi "a" inversand arcele !
  {
    int j;
    color[u]=GRAY;
    ctc[u]=nctc;
    for(j=1;j<=n;j++)
    if((at[u][lista[j]]==1)&&(color[lista[j]]==WHITE)) dfsat(lista[j]);  //"at"
    //if((a[lista[j]][u]==1)&&(color[lista[j]]==WHITE)) dfsat(lista[j]); //"a"
    color[u]=BLACK;
  }
}//class

/*
9 10        1 : 6 7 8
1 2         2 : 9
2 3         3 : 4 5
3 1         4 : 1 2 3
4 5
6 7
7 8
8 9
5 4
7 6
8 7
*/
```

15.3.13 Sortare topologică

Folosind parcurgerea în adâncime

```
// Sortare Topologica = ordonare lineara a varfurilor (in digraf)
// (u,v)=arc ==> "... u ... v ..." in lista ("u se termina inaintea lui v")
// Algoritm:  1. DFS pentru calcul f[u], u=nod
//            2. cand u=terminat ==>  plasaz in lista pe prima pozitie libera
//                                    de la sfarsit catre inceput
// Solutia nu este unica (cea mai mica lexicografic = ???)
// O(n*n)= cu matrice de adiacenta
// O(n+m)= cu liste de adiacenta

import java.io.*;
```

```
class SortTopoDFS
{
  static final int WHITE=0, GRAY=1, BLACK=2;
  static int n,m,t,pozl;   // varfuri, muchii, time, pozitie in lista
  static int[] d;          // descoperit
  static int[] f;          // finalizat
  static int[] color;      // culoare
  static int[] lista;      // lista
  static int[][] a;        // matricea de adiacenta

  public static void main(String[] args) throws IOException
  {
    int i,j,k,nods;         // nods=nod_start_DFS
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("sortTopo.in")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    a=new int[n+1][n+1]; d=new int[n+1]; f=new int[n+1];
    color=new int[n+1];  lista=new int[n+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=1;
    }

    for(i=1;i<=n;i++) color[i]=WHITE;
    t=0;
    pozl=n;
    for(nods=1;nods<=n;nods++)
      if(color[nods]==WHITE) dfs(nods);

    for(i=1;i<=n;i++) System.out.print(lista[i]+" ");
    System.out.println();
  }//main

  static void dfs(int u)
  {
    int v;
    color[u]=GRAY;
    d[u]=++t;
```

```
    for(v=1;v<=n;v++) // mai bine cu liste de adiacenta ... !!!
      if(a[u][v]==1) // v in Ad(u) !!!
        if(color[v]==WHITE) dfs(v);
    color[u]=BLACK;
    f[u]=++t;
    lista[pozl]=u;
    --pozl;
  }//dfs
}//class

/*
6 4           5 6 3 4 1 2
6 3
1 2
3 4
5 6
*/
```

Folosind gradele interioare

```
// Sortare Topologica = ordonare lineara a varfurilor (in digraf)
// (u,v)=arc ==> "... u ... v ..." in lista ("u se termina inaintea lui v")
// Algoritm: cat_timp exista noduri neplasate in lista
//             1. aleg un nod u cu gi[u]=0 (gi=gradul interior)
//             2. u --> lista (pe cea mai mica pozitie neocupata)
//             3. decrementez toate gi[v], unde (u,v)=arc
// OBS: pentru prima solutie lexicografic: aleg u="cel mai mic" (heap!)
// OBS: Algoritm="stergerea repetata a nodurilor de grad zero"

import java.io.*;
class SortTopoGRAD
{
  static final int WHITE=0, BLACK=1;  // color[u]=BLACK ==> u in lista
  static int n,m,pozl;    // varfuri, muchii, pozitie in lista
  static int[] color;     // culoare
  static int[] lista;     // lista
  static int[] gi;        // grad interior
  static int[][] a;       // matricea de adiacenta

  public static void main(String[] args) throws IOException
  {
    int u,i,j,k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("sortTopo.in")));
```

```
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    a=new int[n+1][n+1];
    color=new int[n+1];
    lista=new int[n+1];
    gi=new int[n+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=1;
      gi[j]++;
    }

    for(i=1;i<=n;i++) color[i]=WHITE;
    pozl=1;
    for(k=1;k<=n;k++) // pun cate un nod in lista
    {
      u=nodgi0();
      micsorezGrade(u);
      lista[pozl++]=u;
      color[u]=BLACK;
    }

    for(i=1;i<=n;i++) System.out.print(lista[i]+" ");
    System.out.println();
  }//main

  static int nodgi0() // nod cu gradul interior zero
  {
    int v,nod=-1;
    for(v=1;v<=n;v++) // coada cu prioritati (heap) este mai buna !!!
    if(color[v]==WHITE)
      if(gi[v]==0) {nod=v; break;}
    return nod;
  }

  static void micsorezGrade(int u)
  {
    int v;
    for(v=1;v<=n;v++) // lista de adiacenta este mai buna !!!
```

```
    if(color[v]==WHITE)
      if(a[u][v]==1) gi[v]--;
  }
}//class

/*
6 4          1 2 5 6 3 4
6 3
1 2
3 4
5 6
*/
```

15.3.14 Determinarea nodurilor de separare

```
import java.io.*;      // determinarea nodurilor care strica conexitatea
class NoduriSeparare   // in graf neorientat conex
{
  static int n,m;
  static int [] cc;
  static int[][] a;

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("noduriSeparare.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("noduriSeparare.out")));
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    a=new int[n+1][n+1];
    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      a[i][j]=a[j][i]=1;
    }
    for(i=1;i<=n;i++) if(!econex(i)) System.out.print(i+" ");
    out.close();
  }//main

  static boolean econex(int nodscos)
  {
```

```
    int i, ncc=0;
    int[] cc=new int[n+1];

    for(i=1;i<=n;i++)
      if(i!=nodscos)
        if(cc[i]==0)
        {
          ncc++;
          if(ncc>1) break;
          conex(i,ncc,cc,nodscos);
        }
    if(ncc>1) return false; else return true;
  }// econex()

  static void conex(int u,int et,int[]cc,int nodscos)
  {
    cc[u]=et;
    for(int v=1;v<=n;v++)
      if(v!=nodscos)
        if((a[u][v]==1)&&(cc[v]==0)) conex(v,et,cc,nodscos);
  }//conex
}//class
```

15.3.15 Determinarea muchiilor de rupere

```
import java.io.*;    // determinarea muchiilor care strica conexitatea
class MuchieRupere   // in graf neorientat conex
{
  static int n,m;  static int [] cc;  static int[][]a;

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("muchieRupere.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("muchieRupere.out")));
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    a=new int[n+1][n+1];
    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
```

```
      a[i][j]=1;  a[j][i]=1;
    }
    for(i=1;i<=n;i++)
      for(j=i+1;j<=n;j++)
      {
        if(a[i][j]==0) continue;
        a[i][j]=a[j][i]=0;
        if(!econex())  System.out.println(i+" "+j);
        a[i][j]=a[j][i]=1;
      }
    out.close();
  }//main

  static boolean econex()
  {
    int i, ncc;
    cc=new int[n+1];
    ncc=0;
    for(i=1;i<=n;i++)
    {
      if(cc[i]==0)
      {
        ncc++;
        if(ncc>1) break;
        conex(i,ncc);
      }
    }
    if(ncc==1) return true; else return false;
  }// econex()

  static void conex(int u,int et)
  {
    cc[u]=et;
    for(int v=1;v<=n;v++)
      if((a[u][v]==1)&&(cc[v]==0))
        conex(v,et);
  }//conex
}//class

/*
9 10        1 8
7 2         2 7
5 1         3 9
1 8         7 9
```

```
9 4
6 9
6 4
4 1
9 5
9 7
9 3
*/
```

15.3.16 Determinarea componentelor biconexe

```
// Componenta biconexa = componenta conexa maximala fara muchii de rupere
import java.io.*;         // noduri = 1,...,n
class Biconex             // liste de adiacenta pentru graf
{
  // vs=varf stiva; m=muchii; ncb=nr componente biconexe
  // ndr=nr descendenti radacina (in arbore DFS), t=time in parcurgerea DFS

  static final int WHITE=0, GRAY=1,BLACK=2;
  static int n,ncb,t,ndr,vs,m=0,root;  // root=radacina arborelui DFS
  static int[][] G;          // liste de adiacenta
  static int[] grad,low,d;   // grad nod, low[], d[u]=moment descoperire nod u
  static int[][] B;          // componente biconexe
  static int[] A;            // puncte de articulare
  static int[] color;        // culoarea nodului
  static int[] fs,ts;        // fs=fiu stiva; ts=tata stiva

  public static void main(String[] args) throws IOException
  {
    init();
    root=3;        // radacina arborelui (de unde declansez DFS)
    vs=0;          // pozitia varfului stivei unde este deja incarcat un nod (root)
    fs[vs]=root;   // pun in stiva "root" si
    ts[vs]=0;      // tata_root=0 (nu are!)
    t=0;           // initializare time; numerotarea nodurilor in DFS

    dfs(root,0);   // (u,tatau) tatau=0 ==> nu exista tatau
    if(ncb==1) System.out.println("Graful este Biconex");
    else
    {
      System.out.println("Graful NU este Biconex");
      if(ndr>1) A[root]=1;
      System.out.print("Puncte de articulare : ");
      afisv(A);
```

```
      System.out.print("Numar componente Biconexe : ");
      System.out.println(ncb);
      for(int i=1;i<=ncb;i++)
      {
        System.out.print("Componenta Biconexa "+i+" : ");
        afisv(B[i]);
      }
    }
  }//main()

  static int minim(int a, int b) { return a<b?a:b; } // minim()

  static void init() throws IOException
  {
    int i,j,k;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("biconex.in")));
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    d=new int[n+1];   // vectorii sunt initializati cu zero
    low=new int[n+1]; grad=new int[n+1];   color=new int[n+1]; // 0=WHITE  !
    A=new int[n+1];   G=new int[n+1][n+1]; B=new int[n+1][n+1];
    fs=new int[m+1];  ts=new int[m+1];

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      G[i][++grad[i]]=j; G[j][++grad[j]]=i;
    }
  }//Init()

  static void dfs(int u, int tatau)   /* calculeaza d si low */
  {
    int fiuu,i;

    d[u]=++t;
    color[u]=GRAY;
    low[u]=d[u];

    for(i=1;i<=grad[u];i++)
    {
      fiuu=G[u][i];                  // fiuu = un descendent al lui u
```

```
      if(fiuu==tatau) continue;      // este aceeasi muchie
      if((color[fiuu]==WHITE)||      // fiuu  nedescoperit sau
         (d[fiuu]<d[u]))             // (u,fiuu) este muchie de intoarcere
      {
        /* insereaza in stiva muchia (u,fiuu) */
        vs++;
        fs[vs]=fiuu;
        ts[vs]=u;
      }
      if(color[fiuu]==WHITE)     /* fiuu nu a mai fost vizitat */
      {
        if(u==root) ndr++;       // root=caz special (maresc nrfiiroot)

        dfs(fiuu,u);

           // acum este terminat tot subarborele cu radacina fiuu !!!
        low[u]=minim(low[u],low[fiuu]);
        if(low[fiuu]>=d[u])
           // "=" ==> fiuu intors in u ==> ciclu "agatat" in u !!!
           // ">" ==> fiuu nu are drum de rezerva !!!
        {
          /* u este un punct de articulatie; am identificat o componenta
             biconexa ce contine muchiile din stiva pana la (u,fiuu) */
          if(low[fiuu]!=low[u])               // (u,fiuu) =  bridge (pod)
            System.out.println("Bridge: "+fiuu+" "+u);
          if(u!=root) A[u]=1;                // root = caz special
          compBiconexa(fiuu,u);
        }
      }
      else // (u,fiuu) = back edge
        low[u]=minim(low[u],d[fiuu]);
    }
    color[u]=BLACK;
  } // dfs(...)

  static void compBiconexa(int fiu, int tata)
  {
    int tatas,fius;
    ncb++;
    do
    {
      tatas=ts[vs];    fius=fs[vs];
      vs--;
      B[ncb][tatas]=1; B[ncb][fius]=1;
```

```
    } while(!((tata==tatas)&&(fiu==fius)));
  } // compBiconexa(...)

  static void afisv(int[] x)  // indicii i pentru care x[i]=1;
  {
    for(int i=1;i<=n;i++) if(x[i]==1) System.out.print(i+" ");
    System.out.println();
  }// afisv(...)
}//class

/*
8 9 <-- n m                Bridge: 8 1
1 8                8       Bridge: 5 3
1 2                |       Graful NU este Biconex
1 3   6            1       Puncte de articulare : 1 3 5
3 4   | \        /   \     Numar componente Biconexe : 4
2 4   |  5 ---  3     2    Componenta Biconexa 1 : 1 8
3 5   | /        \   /     Componenta Biconexa 2 : 1 2 3 4
5 7   7            4       Componenta Biconexa 3 : 5 6 7
5 6                        Componenta Biconexa 4 : 3 5
6 7
*/
```

15.3.17 Triangulaţii - OJI2002 clasa a X-a

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.

Datele de intrare: în fişierul text **triang.in** se află pe prima linie un singur număr natural reprezentând valoarea lui n ($n \leq 11$).

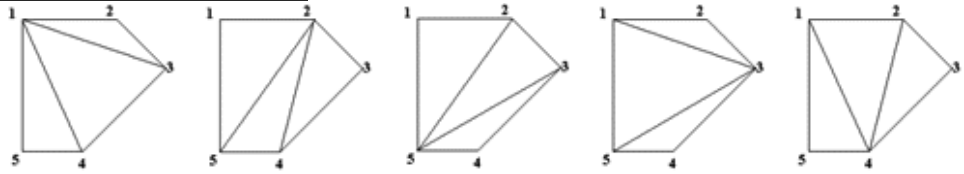
Datele de ieşire: în fişierul text **triang.out** se vor scrie:

- 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	triang.out
5	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.

Rezolvare detaliată

Se genereaza toate combinatiile de diagonale care formeaza o triangulaţie.

```
import java.io.*; // merge si n=12 in 3 sec
class Triangulatii
{
  static int n; // numar varfuri poligon
  static int ndt=n-3; // numar diagonale in triangulatie
  static int nd=n*(n-3)/2; // numarul tuturor diagonalelor
  static int[] x;
  static int[] v1,v2;
  static int nsol=0;
  static PrintWriter out;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("triang.in")));
    out=new PrintWriter(new BufferedWriter(new FileWriter("triang.out")));
    st.nextToken(); n=(int)st.nval;
    ndt=n-3;
    nd=n*(n-3)/2;
    x=new int[ndt+1];
    v1=new int[nd+1];
```

```
    v2=new int[nd+1];

    if(n==3) out.println(0);
    else
    {
      out.println(catalan(n-2));
      diagonale();
      f(1);
    }
    out.close();
    t2=System.currentTimeMillis();
    System.out.println("nsol = "+nsol+" Timp = "+(t2-t1));
  }

  static void afisd() throws IOException
  {
    int i;
    ++nsol;
    for(i=1;i<=ndt;i++) out.print(v1[x[i]]+" "+v2[x[i]]+" ");
    out.println();
  }

  static void diagonale()
  {
    int i,j,k=0;
    i=1;
    for(j=3;j<=n-1;j++) {v1[++k]=i; v2[k]=j;}

    for(i=2;i<=n-2;i++)
    for(j=i+2;j<=n;j++){v1[++k]=i; v2[k]=j;}
  }

  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) throws IOException
  {
```

```
    int i;
    for(i=x[k-1]+1; i<=nd-ndt+k; i++)
    {
      if(seIntersecteaza(k,i)) continue;
      x[k]=i;
      if(k<ndt) f(k+1); else afisd();
    }
  }

  static int catalan(int n)
  {
    int rez;
    int i,j;
    int 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=1;
    for(i=2;i<=n;i++) 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;
  }
}// class
```

15.3.18 Partiţie - ONI2003 clasa a X-a

Se defineşte o partiţie a unui număr natural n ca fiind o scriere a lui n sub forma:

$$n = n_1 + n_2 + ... + n_k, \qquad (k \geq 1)$$

unde $n_1, n_2, ..., nk$ 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.

Datele de intrare

Fişierul **partitie.in** conţine pe prima linie numărul n

Datele de ieşire

Fişerul **partitie.out** va conţine pe prima linie numărul de partiţii ale lui n conform cerinţelor problemei.

Restricţii şi precizări

• $1 \leq N \leq 160$

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

Timp maxim de executare: 3 secunde/test

Indicaţii de rezolvare *

Stelian Ciurea

Problema se poate rezolva în mai multe moduri:

Soluţia comisiei se bazează pe următoarele formule şi teoreme de combinatorică:

– numărul de partiţii ale unui număr n în k părţi (nu neapărat distincte) este

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

cu $P(n,1) = P(n,n) = 1$ şi $P(n,k) = 0$ dacă $n < k$.

– numărul de partiţii ale unui numă n în k părţi distincte este

$$P(n,k) = P(n-k(k-1)/2,k)$$

– numărul de partiţii ale unui număr n în k părţi impare care se pot şi repeta este egal cu numărul de partiţii ale unui număr n în k părţi distincte.

Problema se poate rezolva şi prin backtracking; fără prea mari optimizări se poate obţine rezultatul în mai puţin de 3 secunde pentru $n < 120$. Pentru valori mai mari ale lui n, se poate lăsa programul să ruleze şi se reţin rezultatele într-un vector cu valori iniţiale.

Rezultatul pentru $n = 160$ are 8 cifre, deci nu este necesară implementarea operaţiilor cu numere mari!

Mihai Stroe, GInfo nr. 13/6 - octombrie 2003

Problema se poate rezolva în mai multe moduri.

O idee bună de rezolvare a problemei const în folosirea *metodei programării dinamice*. Se poate construi o matrice A, unde $A_{i,k}$ reprezintă numărul de partiţii ale numărului i cu numere impare, din care ultimul este cel mult egal cu k. Un element din A se calculează observând că o partiţie a lui i cu numere impare, cel mult egale cu k, este formată dintr-un un număr M, mai mic sau egal cu k, şi alte numere, mai mici sau egale cu M. De aici rezultă relaţia:

$$A_{i,k} = \sum_{\substack{M=1,\ldots,k; M \le i; \\ M=impar}} A_{i-M,M}$$

Iniţial $A_{0,0}$ este 1. La implementare, pentru a economisi spaţiu, se poate alege o variantă în care $A_{i,k}$ să reprezinte numărul de partiţii ale lui i cu numere impare, din care ultimul este cel mult egal cu al k-lea număr impar, adică $2 \cdot k - 1$.

După calcularea elementelor matricei, soluţia pentru număul i se găseşte în $A_{i,i}$. Aceste valori pot fi salvate întrun vector de constante, care este transformat într-un nou program, în acelaşi mod ca la problema "Circular" de la clasa a IX-a. Această metodă conduce la o rezolvare instantanee a testelor date în concurs.

O altă metodă, bazată pe vectorul de constante, ar fi însemnat generarea soluţiilor folosind *metoda backtracking*. Un backtracking lăsat să se execute în timpul concursului ar fi putut genera soluţiile pentru toate valorile lui N, după care se putea folosi metoda vectorului de constante, în timp ce un backtracking folosit ca soluţie a problemei ar fi obinut punctajul maxim doar pentru testele mici şi medii (pentru valori ale lui N mai mici decât 130).

Limitele problemei şi timpul de execuie de 3 secunde permit rezolvării prin backtracking ş bţinerea unui punctaj mulţumitor.

Analiza complexităţii

Pentru o rezolvare care se bazează pe metoda vectorului de constante, ordinul de complexitate al soluţiei finale ar fi fost $O(1)$; soluţia constă înn citirea valorii lui N şi afişarea rezultatului memorat.

Soluţia descrisă anterior, bazată pe metoda programării dinamice, are ordinul de complexitate $O(n^3)$. Invităm cititorul să caute metode mai eficiente.

Ordinul de complexitate al unei soluţii care se bazează pe metoda backtracking este exponenţial.

Codul sursă

```
class PartitieNrGenerare // n = suma de numere
{
  static int dim=0,nsol=0;
  static int 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");
  }// main(...)

  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 i;
    int maxok=min(maxp,val);
    for(i=maxok;i>=1;i--)
    {
      x[poz]=i;
      f(val-i,min(val-i,i),poz+1);
    }
  }// f(...)

  static void afis1()
  {
    int i;
    System.out.print("\n"+nsol+" : ");
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }// afis1()

  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 ");
  }afis2(...)

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

class PartitieNrImpare1    // n = suma de numere impare
{                          // nsol = 38328320 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;
    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)+"\n");
  }// main(...)
```

```
  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);
    int i;
    for(i=maxiok;i>=1;i=i-2)
    {
      x[poz]=i;
      f(val-i,min(maxiok,i),poz+1);
    }
  }// f(...)

  static void afis1()
  {
    int i;
    System.out.print("\n"+nsol+" : ");
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }// afis1()

  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 ");
  }// afis2(...)

  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

import java.io.*;
class PartitieNrImpare2   // n = suma de numere impare ;
{                         // nsol = 38328320 timp = 4787
  static int dim=0,nsol=0;
  static int[] x=new int[161];

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int n,i;
    StreamTokenizer st=new StreamTokenizer(
                new BufferedReader(new FileReader("partitie.in")));
    PrintWriter out=new PrintWriter(new BufferedWriter(
                new FileWriter("partitie.out")));

    st.nextToken(); n=(int)st.nval;

    int maxi=((n-1)/2)*2+1;
    f(n,maxi,1);

    out.print(nsol);
    out.close();
    t2=System.currentTimeMillis();
    System.out.println("nsol = "+nsol+" timp= "+(t2-t1));
  }// main(...)

  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);
}// f(...)

  static void afis1()
  {
    int i;
    System.out.print("\n"+nsol+" : ");
    for(i=1;i<=dim;i++) System.out.print(x[i]+" ");
  }// afis1()

  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 ");
  }// afis2(...)

  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
```

15.3.19 Scufiţa - ONI2003 clasa a X-a

Majoritatea participanţilor la ONI2003 au auzit, în copilărie, povestea Scufiţei Roşii. Pentru cei care o ştiu, urmează partea a doua; pentru cei care nu o ştiu, nu vă faceţi griji, cunoaşterea ei nu este necesară pentru a rezolva această problemă. Povestea nu spune ce s-a întâmplat pe drumul pe care Scufiţa Roşie s-a întors de

la bunicuţă. Veţi afla amănunte în continuare.

Pe drum, ea s-a întâlnit cu Lupul (fratele lupului care a părăsit povestea în prima parte). Acesta dorea să o mănânce, dar a decis să-i acorde o şansă de scăpare, provocând-o la un concurs de cules ciupercuţe.

Scufiţa Roşie se află în poziţia $(1, 1)$ a unei matrice cu N linii şi N coloane, în fiecare poziţie a matricei fiind amplasate ciupercuţe. Lupul se află în poziţia $(1, 1)$ a unei alte matrice similare. Pe parcursul unui minut, atât Scufiţa, cât şi Lupul se deplasează într-una din poziţiile vecine (pe linie sau pe coloană) şi culeg ciupercuţele din poziţia respectivă. Dacă Scufiţa Roşie ajunge într-o poziţie în care nu sunt ciupercuţe, va pierde jocul. Dacă la sfârşitul unui minut ea are mai puţine ciupercuţe decât Lupul, ea va pierde jocul de asemenea. Jocul începe după ce amândoi participanţii au cules ciupercuţele din poziţiile lor iniţiale (nu contează cine are mai multe la începutul jocului, ci doar după un număr întreg strict pozitiv de minute de la început). Dacă Scufiţa Roşie pierde jocul, Lupul o va mânca.

Înainte de începerea jocului, Scufiţa Roşie l-a sunat pe Vânător, care i-a promis că va veni într-un sfert de ora (15 minute) pentru a o salva. Deci Scufiţa Roşie va fi liberă să plece dacă nu va pierde jocul după 15 minute.

Din acest moment, scopul ei este nu numai să rămână în viaţă, ci şi să culeagă cât mai multe ciupercuţe, pentru a le duce acasă (după ce va veni, vânătorul nu o va mai lăsa să culeagă).

Lupul, cunoscut pentru lăcomia sa proverbială, va alege la fiecare minut mutarea în câmpul vecin cu cele mai multe ciupercuţe (matricea sa este dată astfel înct să nu existe mai multe posibilităţi de alegere la un moment dat).

Povestea spune că Scufiţa Roşie a plecat acasă cu coşuleţul plin de ciupercuţe, folosind indicaţiile date de un program scris de un concurent la ONI 2003 (nu vom da detalii suplimentare despre alte aspecte, cum ar fi călătoria în timp, pentru a nu complica inutil enunţul problemei). Să fi fost acest program scris de către dumneavoastră? Vom vedea...

Cerinţă

Scrieţi un program care să o ajute pe Scufiţa Roşie să rămână în joc şi să culeagă cât mai multe ciupercuţe la sfârşitul celor 15 minute!

Date de intrare

Fişierul **scufita.in** are următoarea structură:

N - dimensiunea celor două matrice

$a_{11}a_{12}...a_{1n}$ -matricea din care culege Scufiţa Roşie

$a_{21}a_{22}...a_{2n}$

...

$a_{n1}a_{n2}...a_{nn}$

$b_{11}b_{12}...b_{1n}$ - matricea din care culege Lupul

$b_{21}b_{22}...b_{2n}$

...

$b_{n1}b_{n2}...b_{nn}$

Date de ieşire

Fişierul **scufita.out** are următoarea structură:

NR - numărul total de ciupercuţe culese

$d_1d_2...d_{15}$ - direcţiile pe care s-a deplasat Scufiţa Roşie, separate prin câte un spaţiu (direcţiile pot fi N, E, S, V indicând deplasări spre Nord, Est, Sud, Vest; poziţia $(1, 1)$ este situată în colţul de Nord-Vest al matricei)

Restricţii

• $4 \leq N \leq 10$;

• valorile din ambele matrice sunt numere naturale mai mici decât 256;

• nici Scufiţa Roşie şi nici Lupul nu vor părăsi matricele corespunzătoare;

• după ce unul din jucători culege ciupercuţele dintr-o poziţie, în poziţia respectivă rămân 0 ciupercuţe;

• pe testele date, Scufiţa Roşie va avea întotdeauna posibilitatea de a rezista 15 minute.

Exemplu

scufita.in	**scufita.out**
4	137
2 2 3 4	SSSEEENVVNEENVV
5 6 7 8	
9 10 11 12	
13 14 15 16	
1 2 3 4	
5 6 7 8	
9 10 11 12	
13 14 15 16	

Explicaţie:

Scufiţa Roşie a efectuat aceleaşi mutări cu cele efectuate de Lup şi a avut tot timpul o ciupercuţă în plus. În final ea a cules toate ciupercuţele din matrice.

Timp maxim de executare: 1 secundă/test

Indicaţii de rezolvare *

Mihai Stroe, GInfo nr. 13/6 - octombrie 2003

Problema se rezolvă folosind *metoda backtracking* în plan.

Se observă că mutările *Lupului* sunt independente de mutările *Scufiţei*, astfel ele pot fi determinate imediat după citirea datelor.

În acest punct al rezolvării, *Scufiţa* trebuie să mute la fiecare moment, astfel încât să rămână în joc, iar în final să strângă cât mai multe ciupercuţe. Restricţiile sunt date de enunţul problemei şi de numărul de ciupercuţe culese de *Lup*, care este cunoscut la fiecare moment.

Rezolvarea prin *metoda backtracking* încearcă la fiecare pas, pe rând, fiecare mutare din cele cel mult patru posibile (teoretic; de fapt, cel mult trei mutări sunt posibile în fiecare moment, deoarece Scufiţa nu se va întoarce în poziţia din care tocmai a venit).

Se urmăreşte respectarea restricţiilor impuse. În momentul găsirii unei soluţii, aceasta este comparată cu soluţia optimă găsită până atunci şi dacă este mai bună este reţinută.

Încă nu a fost găsită o rezolvare polinomială pentru această problemă (şi este improbabil ca o astfel de rezolvare să existe), dar limitele mici şi faptul că numărul de mutări disponibile începe să scadă destul de repede, în multe cazuri permit un timp de execuţie foarte bun.

Analiza complexităţii

Fie M numărul de mutări pe care le are de efectuat Scufiţa.

Operaţiile de citire şi scriere a datelor au ordinul de complexitate $O(N^2)$, respectiv $O(M)$, deci nu vor fi luate în calcul. Ordinul de complexitate este dat de rezolvarea prin *metoda backtracking*.

Având în vedere că la fiecare pas *Scufiţa* poate alege dintre cel mult 3 mutări (deoarece nu se poate întoarce în poziţia din care a venit), ordinul de complexitate ar fi $O(3^M)$. De fapt, numărul maxim de stări examinate este mult mai mic; de exemplu, primele două mutări oferă două variante de alegere în loc de trei. Alte restricţii apar datorită limitării la o matrice de $10 \cdot 10$ elemente.

Cum numărul M este cunoscut şi mic, s-ar putea considera că ordinul de complexitate este limitat superior, deci constant (constanta introdusă fiind totuşi destul de mare).

Codul sursă

```
import java.io.*;
class Scufita
{
  static int n, maxc=0;
  static int[][] a,b;
  static int[] x,y;
  static char[] tc=new char[16];
  static char[] tmax=new char[16];

  public static void main(String[] args) throws IOException
  {
    int i,j;
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("scufita.in")));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("scufita.out")));

    st.nextToken(); n=(int)st.nval;
```

```
    a=new int[n+2][n+2];
    b=new int[n+2][n+2];
    x=new int[17];
    y=new int[17];

    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++) { st.nextToken(); a[i][j]=(int)st.nval; }
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++) { st.nextToken(); b[i][j]=(int)st.nval; }

    culegeLupul(1,1,1);
    f(1,1,1);

    out.println(maxc);
    for(i=1;i<=15;i++) out.print(tmax[i]);
    out.println();
    out.close();
  }// main()

  static void f(int i, int j, int k) throws IOException
  {
    int aij=a[i][j];

    x[k]=x[k-1]+a[i][j];
    a[i][j]=0;

    if(k==16)
    {
      if(x[16]>maxc)
      {
        maxc=x[16];
        for(int ii=1;ii<=15;ii++) tmax[ii]=tc[ii];
      }
      a[i][j]=aij;
      return;
    }

    if((a[i-1][j]>0)&&(a[i-1][j]+x[k]>=y[k+1])) { tc[k]='N'; f(i-1,j,k+1); }
    if((a[i+1][j]>0)&&(a[i+1][j]+x[k]>=y[k+1])) { tc[k]='S'; f(i+1,j,k+1); }
    if((a[i][j-1]>0)&&(a[i][j-1]+x[k]>=y[k+1])) { tc[k]='V'; f(i,j-1,k+1); }
    if((a[i][j+1]>0)&&(a[i][j+1]+x[k]>=y[k+1])) { tc[k]='E'; f(i,j+1,k+1); }

    a[i][j]=aij;
  }//f(...)
```

```
  static void culegeLupul(int i, int j, int k)
  {
    if(k>16) return;
    y[k]=y[k-1]+b[i][j];
    b[i][j]=0;
    if((b[i-1][j]>b[i+1][j])&&(b[i-1][j]>b[i][j-1])&&(b[i-1][j]>b[i][j+1]))
      culegeLupul(i-1,j,k+1);
    else if((b[i+1][j]>b[i][j-1])&&(b[i+1][j]>b[i][j+1]))
      culegeLupul(i+1,j,k+1);
    else if(b[i][j-1]>b[i][j+1])
      culegeLupul(i,j-1,k+1);
    else culegeLupul(i,j+1,k+1);
  }// culegeLupul(...)
}// class
```

Capitolul 16

Programare dinamică

16.1 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.

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,

3. calculul efectiv al valorii unei soluţii optime,

4. reconstituirea unei soluţii pe baza informaţiei calculate.

16.2 Probleme rezolvate

16.2.1 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).

```
import java.io.*;
class InmOptimalaMatrice
{
  static int nmax=20;
  static int m[][]=new int[100][100];
  static int p[]=new int[100];
  static int n,i,j,k,imin,min,v;

  public static void paranteze(int i,int j)
  {
    int k;
    if(i<j)
    {
      k=m[j][i];
      if(i!=k)
      {
        System.out.print("(");
        paranteze(i,k);
```

```
      System.out.print(")");
    }//if
    else paranteze(i,k);

    System.out.print(" * ");

    if(k+1!=j)
    {
      System.out.print("(");
      paranteze(k+1,j);
      System.out.print(")");
    }//if
    else paranteze(k+1,j);
  }//if
  else System.out.print("A"+i);
}//paranteze

public static void main(String[]args) throws IOException
{
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

  System.out.print("numarul matricelor: ");
  n=Integer.parseInt(br.readLine());

  System.out.println("Dimensiunile matricelor:");
  for(i=1;i<=n+1;i++)
  {
    System.out.print("p["+i+"]=");
    p[i]=Integer.parseInt(br.readLine());
  }

  for(i=n;i>=1;i--)
    for(j=i+1;j<=n;j++)
    {
      min=m[i][i]+m[i+1][j]+p[i]*p[i+1]*p[j+1];
      imin=i;
      for(k=i+1;k<=j-1;k++)
      {
        v=m[i][k]+m[k+1][j]+p[i]*p[k+1]*p[j+1];
        if(min>v) { min=v; imin=k; }
      }
      m[i][j]=min;
      m[j][i]=imin;
    }//for i,j
```

```
    System.out.println("Numarul minim de inmultiri este: "+m[1][n]);
    System.out.print("Ordinea inmultirilor: ");
    paranteze(1,n);
    System.out.println();
  }//main
}//class
/*
numarul matricelor: 8
Dimensiunile matricelor:
p[1]=2
p[2]=8
p[3]=3
p[4]=2
p[5]=7
p[6]=2
p[7]=5
p[8]=3
p[9]=7
Numarul minim de inmultiri este: 180
Ordinea inmultirilor: ((((A1 * A2) * A3) * (A4 * A5)) * (A6 * A7)) * A8
*/
```

16.2.2 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
```

16.2.3 Sumă maximă în triunghi de numere

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 16.1: 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.

16.2.4 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();
}

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 0
1 0 1 1 1 1 1 1 1 1 1 1 1 1
2 0 1 1 2 2 2 2 2 2 2 2 2 2
3 0 1 2 2 2 2 2 2 2 2 2 2 2
4 0 1 2 2 3 3 3 3 3 3 3 3 3
5 0 1 2 2 3 3 4 4 4 4 4 4 4
6 0 1 2 2 3 4 4 4 4 4 4 4 4
a 0 1 2 2 3 4 4 5 5 5 5 5 5
b 0 1 2 2 3 4 4 5 5 6 6 6 6
c 0 1 2 2 3 4 4 5 6 6 6 6 6
d 0 1 2 2 3 4 4 5 6 6 7 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
```

16.2.5 Distanţa minimă de editare

Fie $d(s1, s2)$ distanţa de editare (definită ca fiind numărul minim de operaţii de *ştergere*, *inserare* şi *modificare*) dintre şirurile de caractere $s1$ şi $s2$. Atunci:

$$d(\text{""}, \text{""}) = 0 \tag{16.2.1}$$

$$d(s, \text{""}) = d(\text{""}, s) = |s|, \tag{16.2.2}$$

Având în vedere ultima operaţie efectuată asupra primului şir de caractere, la sfârşitul acestuia (dar după modificările efectuate deja), obţinem:

$$d(s_1+ch_1, s_2+ch_2) = \min \begin{cases} d(s_1+ch_1, s_2)+1, \text{inserare } ch_2 \\ d(s_1, s_2+ch_2)+1, \text{ştergere } ch_1 \\ d(s_1,s_2) + \begin{cases} 0 & \text{dacă } ch_1 = ch_2, nimic! \\ 1 & \text{dacă } ch_1 \neq ch_2, \text{ înlocuire} \end{cases} \end{cases} \quad (16.2.3)$$

Folosim o matrice $c[0..|s1|][0..|s2|]$ unde $c[i][j] \stackrel{\text{def}}{=} d(s_1[1..i], s_2[1..j])$.
Algoritmul în pseudocod este:

```
m[0][0]=0;
for(i=1; i<=length(s1); i++) m[i][0]=i;
for(j=1; j<=length(s2); j++) m[0][j]=j;

for(i=1; i<=length(s1); i++)
  for(j=1;j<=length(s2); j++)
  {
    val=(s1[i]==s2[j]) ? 0 : 1;
    m[i][j]=min( m[i-1][j-1]+val, m[i-1][j]+1, m[i][j-1]+1 );
  }
```

Programul sursă:

```
import java.io.*;
class DistEdit
{
  static final char inserez='i', // inserez inaintea pozitiei
                    sterg='s',
                    modific='m',
                    nimic=' '; // caracter !!!
  static final char sus='|',
                    stanga='-',
                    diags='x';

  static int na,nb;
  static int [][] c;    // c[i][j] = cost a1..ai --> b1..bj
  static char [][] op;  // op = operatia efectuata
  static char [][] dir; // dir = directii pentru minim !!!

  static String a,b;    // a--> b

  public static void main(String[] args) throws IOException
  {
    int i,j,cmin=0;
```

```
BufferedReader br=new BufferedReader(
  new FileReader("distedit.in"));
PrintWriter out=new PrintWriter(
  new BufferedWriter(new FileWriter("distedit.out")));

a=br.readLine();
b=br.readLine();
na=a.length();
nb=b.length();

c=new int[na+1][nb+1];
op=new char[na+1][nb+1];
dir=new char[na+1][nb+1];

System.out.println(a+" --> "+na);
System.out.println(b+" --> "+nb);

for(i=1;i<=na;i++)
{
  c[i][0]=i;      // stergeri din a; b=vid !!!
  op[i][0]=sterg; // s_i
  dir[i][0]=sus;
}

for(j=1;j<=nb;j++)
{
  c[0][j]=j;        // inserari in a;  a=vid !!!
  op[0][j]=inserez; //t_j
  dir[0][j]=stanga;
}

op[0][0]=nimic;
dir[0][0]=nimic;

for(i=1;i<=na;i++)
{
  for(j=1;j<=nb;j++)
  {
    if(a.charAt(i-1)==b.charAt(j-1))
    {
      c[i][j]=c[i-1][j-1];
      op[i][j]=nimic;
      dir[i][j]=diags;
    }
```

```
        else
        {
          cmin=min(c[i-1][j-1],c[i-1][j],c[i][j-1]);
          c[i][j]=1+cmin;
          if(cmin==c[i][j-1])    // inserez t_j
          {
            op[i][j]=inserez;
            dir[i][j]=stanga;
          }
          else
          if(cmin==c[i-1][j]) // sterg s_i
          {
            op[i][j]=sterg;
            dir[i][j]=sus;
          }
          else
          if(cmin==c[i-1][j-1]) //s_i-->t_j
          {
            op[i][j]=modific;
            dir[i][j]=diags;
          }
        }// else
      }// for j
    }// for i

    afismi(c,out);
    afismc(dir,out);
    afismc(op,out);

    afissol(na,nb,out);
    out.println("\nCOST transformare = "+c[na][nb]);
    out.close();
    System.out.println("COST transformare = "+c[na][nb]);
  }// main

  static void afismc(char[][] x, PrintWriter out)
  {
    int i,j;
    out.print("  ");
    for(j=1;j<=nb;j++) out.print(b.charAt(j-1));
    for(i=0;i<=na;i++)
    {
      out.println();
      if(i>0) out.print(a.charAt(i-1)); else out.print(" ");
```

```
      for(j=0;j<=nb;j++) out.print(x[i][j]);
    }
    out.println("\n");
  }// afismc(...)

  static void afismi(int[][] x, PrintWriter out)
  {
    int i,j;
    out.print("  ");
    for(j=1;j<=nb;j++) out.print(b.charAt(j-1));
    for(i=0;i<=na;i++)
    {
      out.println();
      if(i>0) out.print(a.charAt(i-1)); else out.print(" ");
      for(j=0;j<=nb;j++) out.print(x[i][j]);
    }
    out.println("\n");
  }// afismi(...)

  static void afissol(int i,int j, PrintWriter out)
  {
    if(i==0&&j==0) return;
    if(dir[i][j]==diags) afissol(i-1,j-1,out);
    else
      if(dir[i][j]==stanga) afissol(i,j-1,out);
      else
        if(dir[i][j]==sus) afissol(i-1,j,out);

    if((i>0)&&(j>0))
    {
      if(op[i][j]==sterg)
        out.println(i+" "+a.charAt(i-1)+" "+op[i][j]);
      else
      if(op[i][j]==inserez)
        out.println("    "+op[i][j]+" "+j+" "+b.charAt(j-1));
      else
        out.println(i+" "+a.charAt(i-1)+" "+op[i][j]+" "+j+" "+b.charAt(j-1));
    }
    else
    if(i==0)
      out.println(i+" "+a.charAt(i)+" "+op[i][j]+" "+j+" "+b.charAt(j-1));
    else
    if(j==0)
      out.println(i+" "+a.charAt(i-1)+" "+op[i][j]+" "+j+" "+b.charAt(j));
```

```
  }//afissol(...)

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

Rezultate afişate în fişierul de ieşire:

```
  altruisti
 0123456789
a1012345678
l2101234567
g3211234567
o4322234567
r5433234567
i6544333456
t7654444445
m8765555555
i9876665665

  altruisti
  ---------
a|x--------
l||x-------
g|||x------
o||||x-----
r||||x-----
i|||||xx--x
t|||x|||xx-
m|||||||||x
i||||||x-|x

  altruisti
  iiiiiiiii
as iiiiiiii
lss iiiiiii
gsssmiiiiii
ossssmiiiii
rssss iiiii
issssm ii
tsss sssm i
mssssssssm
isssss is
```

```
1 a   1 a
2 l   2 l
3 g m 3 t
4 o s
5 r   4 r
    i 5 u
6 i   6 i
    i 7 s
7 t   8 t
8 m s
9 i   9 i

COST transformare = 5
```

16.2.6 Problema rucsacului $(0-1)$

```
import java.io.*;
class Rucsac01
{
  public static void main(String[] args) throws IOException
  {
    int n,m;
    int i,j;
    int[] g,v;
    int[][] c;

    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("rucsac.out")));
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("rucsac.in")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    g=new int[n+1];
    v=new int[n+1];
    c=new int[n+1][m+1];

    for(i=1;i<=n;i++) { st.nextToken(); g[i]=(int)st.nval; }
    for(i=1;i<=n;i++) { st.nextToken(); v[i]=(int)st.nval; }

    for(i=1; i<=n; i++) c[i][0]=0;
```

```
    for(j=0; j<=m; j++) c[0][j]=0;

    for(i=1; i<=n; i++)
      for(j=1; j<=m; j++)
        if(g[i]>j)
          c[i][j]=c[i-1][j];
        else
          c[i][j]=max( c[i-1][j], c[i-1][j-g[i]] + v[i] );

    out.println(c[n][m]);
    out.close();
  }// main(...)

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }// max(...)
}
```

16.2.7 Problema schimbului monetar

```
import java.io.*;
class Schimb
{
  public static void main(String[] args) throws IOException
  {
    int v,n;
    int i,j;
    int[] b, ns;

    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("schimb.out")));
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("schimb.in")));

    st.nextToken(); v=(int)st.nval;
    st.nextToken(); n=(int)st.nval;

    b=new int[n+1];
    ns=new int[v+1];

    for(i=1;i<=n;i++) { st.nextToken(); b[i]=(int)st.nval; }
```

```
    ns[0]=1;
    for(i=1; i<=n; i++)
      for(j=b[i]; j<=v; j++)
        ns[j]+=ns[j-b[i]];
    out.println(ns[v]);
    out.close();
  }// main
}// class
```

16.2.8 Problema traversării matricei

Traversarea unei matrice de la Vest la Est; sunt permise deplasări spre vecinii unei poziţii (chiar şi deplasări prin "exteriorul" matricei).

```
import java.io.*;
class Traversare
{
  public static void main(String[] args) throws IOException
  {
    int m,n;
    int i,j,imin;
    int[][] a,c,t;
    int[] d;
    int cmin,minc;

    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("traversare.out")));
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("traversare.in")));

    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;

    a=new int[m][n];
    c=new int[m][n];
    t=new int[m][n];
    d=new int[n];

    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
      {
        st.nextToken();
```

```
        a[i][j]=(int)st.nval;
      }

    for(i=0;i<m;i++) c[i][0]=a[i][0];
    for(j=1; j<n; j++)
      for(i=0; i<m; i++)
      {
        minc=min(c[(i+m-1)%m][j-1], c[i][j-1], c[(i+1)%m][j-1]);
        c[i][j]=a[i][j]+minc;
        if(minc==c[(i+m-1)%m][j-1]) t[i][j]=((i+m-1)%m)*n+(j-1);
        else
          if(minc==c[i][j-1]) t[i][j]=i*n+(j-1);
          else t[i][j]=((i+1)%m)*n+(j-1);
      }

    imin=0;
    cmin=c[0][n-1];
    for(i=1;i<m;i++)
      if(c[i][n-1]<cmin)
      {
        cmin=c[i][n-1];
        imin=i;
      }
    out.println(cmin);

    d[n-1]=imin*n+(n-1);
    j=n-1;
    i=imin;
    while(j>0)
    {
      i=t[i][j]/n;
      j--;
      d[j]=i*n+j;
    }
    for(j=0;j<n;j++) out.println((1+d[j]/n)+" "+(1+d[j]%n));
    out.close();
}// main(...)

  static int min(int a, int b)
  {
    if(a<b) return a; else return b;
  }

  static int min(int a, int b, int c)
```

```
  {
    return min(min(a,b),c);
  }
}// class
/*
traversare.in     traversare.out
3 4               6
2 1 3 2           2 1
1 3 5 4           1 2
3 4 2 7           3 3
                  1 4
*/
```

16.2.9 Problema segmentării vergelei

Scopul algoritmului este de a realiza n tăieturi, dea lungul unei vergele în locuri pre-specificate, cu efort minim (sau *cost*). Costul fiecărei operaţii de tăiere este proporţional cu lungimea vergelei care trebuie tăiată. În viaţa reală, ne putem imagina *costul* ca fiind *efortul* depus pentru a plasa vergeaua (sau un buştean!) în maşina de tăiat.

Considerăm şirul de numere naturale $0 < x_1 < x_2 < ... < x_n < x_{n+1}$ în care x_{n+1} reprezintă lungimea vergelei iar $x_1, x_2, ..., x_n$ reprezintă abscisele punctelor în care se vor realiza tăieturile (distanţele faţă de capătul "din stânga" al vergelei).

Notăm prin $c[i][j]$ $(i < j)$ *costul minim* necesar realizării tuturor tăieturilor segmentului de vergea $[x_i..x_j]$.

Evident $c[i][i+1] = 0$ pentru că nu este necesară nici o tăietură.

Pentru $j > i$ să presupunem că realizăm prima tăietură în x_k $(i < k < j)$. Din vergeaua $[x_i...x_j]$ obţinem două bucaţi mai mici: $[x_i...x_k]$ şi $[x_k...x_j]$. Costul pentru tăierea vergelei $[x_i...x_j]$ este format din costul transportului acesteia la maşina de tăiat $(x_j - x_i)$ + costul tăierii vergelei $[x_i...x_k]$ (adică $c[i][k]$) şi + costul tăierii vergelei $[x_k...x_j]$ (adică $c[k][j]$).

Dar, ce valoare are k? Evident, k trebuie să aibă acea valoare care să minimizeze expresia $c[i][k] + c[k][j]$. Obţinem:

$$c[i][j] = x_j - x_i + \min_{i<k<j}\{c[i][k] + c[k][j]\}$$

```
import java.io.*;
class Vergea
{
  static int n,nt=0;
  static int x[];
  static int c[][];
  static int t[][];
```

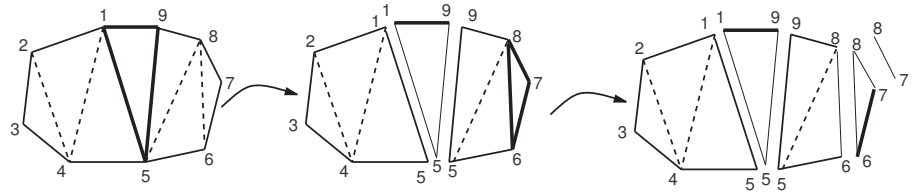
```
static BufferedReader br; // pentru "stop" (pentru depanare!)

public static void main(String[]args) throws IOException
{
  int i,j,h,k,min,kmin;
  StreamTokenizer st=new StreamTokenizer(
    new BufferedReader(new FileReader("vergea.in")));
  PrintWriter out=new PrintWriter(
    new BufferedWriter(new FileWriter("vergea.out")));
  br=new BufferedReader(new InputStreamReader(System.in));

  st.nextToken(); n=(int)st.nval;
  x=new int[n+2];
  c=new int[n+2][n+2];
  t=new int[n+2][n+2];

  for(i=1;i<=n+1;i++) { st.nextToken(); x[i]=(int)st.nval; }

  System.out.println("n="+n);
  System.out.print("x: ");
  for(i=1;i<=n+1;i++) System.out.print(x[i]+" ");
  System.out.println();

  for(i=0;i<=n;i++) c[i][i+1]=0;

  j=-1;
  for(h=2;h<=n+1;h++)        // lungimea vargelei
  {
    for(i=0;i<=n+1-h;i++)    // inceputul vargelei
    {
      j=i+h;                 // sfarsitul vargelei
      c[i][j]=x[j]-x[i];
      min=Integer.MAX_VALUE;
      kmin=-1;
      for(k=i+1;k<=j-1;k++)
        if(c[i][k]+c[k][j]<min)
        {
          min=c[i][k]+c[k][j];
          kmin=k;
        }
      c[i][j]+=min;
      t[i][j]=kmin;
    }//for i
```

```
    }// for h

    out.println(c[0][n+1]);
    out.close();

    afism(c); afism(t);
    System.out.println("Ordinea taieturilor: \n");
    taieturi(0,n+1);
    System.out.println();
  }//main

  public static void taieturi(int i,int j) throws IOException
  {
    if(i>=j-1) return;
    int k;
    k=t[i][j];
    System.out.println((++nt)+" : "+i+".."+j+" --> "+k);
    //br.readLine();                  // "stop" pentru depanare !
    if((i<k)&&(k<j)) { taieturi(i,k); taieturi(k,j); }
  }//taieturi

  static void afism(int[][] x)       // pentru depanare !
  {
    int i,j;
    for(i=0;i<=n+1;i++)
    {
      for(j=0;j<=n+1;j++) System.out.print(x[i][j]+" ");
      System.out.println();
    }
    System.out.println();
  }// afism(...)
}//class
/*
n=5
x: 2 4 5 8 12 15

0 0 4 8 16 27 38
0 0 0 3  9 19 29
0 0 0 0  4 12 22
0 0 0 0  0  7 17
0 0 0 0  0  0  7
0 0 0 0  0  0  0
0 0 0 0  0  0  0
```

```
0 0 1 1 2 3 4
0 0 0 2 3 4 4
0 0 0 0 3 4 4
0 0 0 0 0 4 4
0 0 0 0 0 0 5
0 0 0 0 0 0 0
0 0 0 0 0 0 0

Ordinea taieturilor:

1 : 0..6 --> 4
2 : 0..4 --> 2
3 : 0..2 --> 1
4 : 2..4 --> 3
5 : 4..6 --> 5
*/
```

16.2.10 Triangularizarea poligoanelor convexe

Considerăm un poligon convex cu n vârfuri numerotate cu 1, 2, ..., n (în figură $n = 9$) şi dorim să obţinem o triangularizare în care suma lungimilor diagonalelor trasate să fie minimă (ca şi cum am dori să consumăm cât mai puţin tuş pentru trasarea acestora!).



Evident, orice latură a poligonului face parte dintr-un triunghi al triangulaţiei. Considerăm la început latura $[1, 9]$. Să presupunem că într-o anumită triangulaţie optimă latura $[1, 9]$ face parte din triunghiul $[1, 5, 9]$. Diagonalele triangulaţiei optime vor genera o triangulaţie optimă a poligoanelor convexe $[1, 2, 3, 4, 5]$ şi $[5, 6, 7, 8, 9]$. Au apărut astfel două subprobleme ale problemei iniţiale.

Să notăm prin $p(i, k, j)$ perimetrul triunghiului $[i, k, j]$ $(i < k < j)$ işi prin $c[i][j]$ costul minim al triagulaţiei poligonului convex $[i, i + 1, ..., j]$ (unde $i < j$). Atunci:

$$c[i][j] = 0, \text{dacă } j = i + 1$$

şi

$$c[i][j] = \min_{i \le k < j} \{p(i,k,j) + c[i][k] + c[k][j]\}, \text{dacă } i < j \le n$$

16.2.11 Algoritmul Roy-Floyd-Warshall

```
// Lungime drum minim intre oricare doua varfuri in graf orientat ponderat.
// OBS: daca avem un drum de lungime minima de la i la j atunci acest drum
//  va trece numai prin varfuri distincte, iar daca varful k este varf intermediar,
//  atunci drumul de la i la k si drumul de la k la j sunt si ele minime
//  (altfel ar exista un drum mai scurt de la i la j); astfel, este indeplinit
// "principiul optimalitatii"

import java.io.*;
class RoyFloydWarshall            // O(n^3)
{
  static final int oo=0x7fffffff;
  static int n,m;
  static int[][] d;

  public static void main(String[]args) throws IOException
  {
    int i,j,k;

    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("RoyFloydWarshall.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("RoyFloydWarshall.out")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    d=new int[n+1][n+1];

    for(i=1;i<=n;i++) for(j=1;j<=n;j++) d[i][j]=oo;
    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); d[i][j]=d[j][i]=(int)st.nval;
    }

    for(k=1;k<=n;k++)
      for(i=1;i<=n;i++)      // drumuri intre i si j care trec
```

```
        for(j=1;j<=n;j++)    // numai prin nodurile 1, 2, ..., k
          if((d[i][k]<oo)&&(d[k][j]<oo))
            if(d[i][j]>d[i][k]+d[k][j]) d[i][j]=d[i][k]+d[k][j];

    for(i=1;i<=n;i++)
    {
      for(j=1;j<=n;j++)
        if(d[i][j]<oo) System.out.print(d[i][j]+" ");
        else System.out.print("*"+" ");
      System.out.println();
    }
    out.close();
  }//main
}//class

/*
6 6
1 2 3     2 3 1 * * *
1 3 1     3 4 2 * * *
2 3 2     1 2 2 * * *
4 5 1     * * * 2 1 2
5 6 3     * * * 1 2 3
4 6 2     * * * 2 3 4
*/
```

16.2.12 Oracolul decide - ONI2001 cls 10

prof. Doru Popescu Anastasiu, Slatina

La un concurs participă N concurenţi. Fiecare concurent primeşte o foaie de hârtie pe care va scrie un cuvânt având cel mult 100 de caractere (litere mici ale alfabetului englez). Cuvintele vor fi distincte.

Pentru departajare, concurenţii apelează la un oracol. Acesta produce şi el un cuvnt. Va câştiga concurentul care a scris cuvântul "cel mai apropiat" de al oracolului.

Gradul de "apropiere" dintre două cuvinte este lungimea subcuvântului comun de lungime maximă. Prin subcuvânt al unui cuvânt dat se înţelege un cuvânt care se poate obţine din cuvântul dat, eliminând 0 sau mai multe litere şi păstrând ordinea literelor rămase.

Cerinţă

Se cunosc cuvântul c_0 produs de oracol şi cuvintele c_i, $i = 1, ..., N$ scrise de concurenţi. Pentru a ajuta comisia să desemneze câştigătorul, se cere ca pentru fiecare i să identificaţi poziţiile literelor ce trebuie şterse din c_0 şi din c_i astfel încât prin ştergere să se obţină unul dintre subcuvintele comune de lungime maximă.

Date de intrare

Fişier de intrare: ORACOL.IN

Linia 1: N număr natural nenul, reprezentând numărul concurenţilor;

Linia 2: c_0 cuvântul produs de oracol;

Liniile 3..N+2: *cuvânt* pe aceste N linii se află cuvintele scrise de cei N concurenţi, un cuvânt pe o linie;

Date de ieşire

Fişier de ieşire: ORACOL.OUT

Liniile 1..2*N: *poziţiile literelor ce trebuie şterse* pe fiecare linie i ($i = 1, 3, ..., 2 * N - 1$) se vor scrie numere naturale nenule, separate prin câte un spaţiu, reprezentând poziţiile de pe care se vor şterge litere din cuvântul produs de oracol; pe fiecare linie j ($j = 2, 4, ..., 2 * N$) se vor scrie numere naturale nenule, separate prin câte un spaţiu, reprezentând poziţiile de pe care se vor şterge litere din cuvântul concurentului cu numărul $j/2$.

Restricţii

$2 \leq N \leq 100$

Dacă există mai multe soluţii, în fişier se va scrie una singură.

Dacă dintr-un cuvânt nu se va tăia nici o literă, linia respectivă din fişierul de intrare va rămâne vidă.

Exemplu

ORACOL.IN	ORACOL.OUT poate conţine soluţia:
3	3
abc	3 4
abxd	
aabxyc	1 4 5
acb	3
	2

Timp maxim de executare/test: 1 secundă

Codul sursă

Varianta 1:

```
import java.io.*;  // subsir comun maximal - problema clasica ...
class Oracol       // varianta cu mesaje pentru depanare ...
{
  static final char sus='|', stanga='-', diag='*';
  static int[][] a;
  static char[][] d;

  static String x,y;
  static boolean[] xx=new boolean[101];
  static boolean[] yy=new boolean[101];
```

```
  static char[] z;
  static int m,n,nc;

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    BufferedReader br=new BufferedReader(new FileReader("oracol.in"));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("oracol.out")));

    nc=Integer.parseInt(br.readLine());

    x=br.readLine().replace(" ",""); // elimina spatiile ... de la sfarsit !
    m=x.length();

    for(k=1;k<=nc;k++)
    {
      y=br.readLine().replaceAll(" ",""); // elimina spatiile ... daca sunt!
      n=y.length();

      matrad();
      afism(a);
      afism(d);

      System.out.print("O solutie oarecare: ");
      z=new char[a[m][n]+1];
      for(i=1;i<=m;i++) xx[i]=false;
      for(j=1;j<=n;j++) yy[j]=false;
      osol(m,n);
      System.out.println("\n");

      for(i=1;i<=m;i++) if(!xx[i]) out.print(i+" ");
      out.println();
      for(j=1;j<=n;j++) if(!yy[j]) out.print(j+" ");
      out.println();
    }
    out.close();
    System.out.println("\n");
  }// main(...)

  static void matrad()
  {
    int i,j;
    a=new int[m+1][n+1];
```

```
    d=new char[m+1][n+1];
    for(i=1;i<=m;i++)
      for(j=1;j<=n;j++)
        if(x.charAt(i-1)==y.charAt(j-1))
        {
          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;
        }
  }// matrad()

  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));
      xx[lin]=yy[col]=true;
    }
  }// osol(...)

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }// max(...)

  static void afism(int[][] a)
  {
    int i,j;

    System.out.print("    ");
    for(j=0;j<n;j++) System.out.print(y.charAt(j)+" ");
    System.out.println();
```

```
    System.out.print("  ");
    for(j=0;j<=n;j++) System.out.print(a[0][j]+" ");
    System.out.println();

    for(i=1;i<=m;i++)
    {
      System.out.print(x.charAt(i-1)+" ");
      for(j=0;j<=n;j++) System.out.print(a[i][j]+" ");
      System.out.println();
    }
    System.out.println("\n");
  }// afism(int[][]...)

  static void afism(char[][] d) // difera tipul parametrului
  {
    int i,j;
    System.out.print("    ");
    for(j=0;j<n;j++) System.out.print(y.charAt(j)+" ");
    System.out.println();

    System.out.print("  ");
    for(j=0;j<=n;j++) System.out.print(d[0][j]+" ");
    System.out.println();

    for(i=1;i<=m;i++)
    {
      System.out.print(x.charAt(i-1)+" ");
      for(j=0;j<=n;j++) System.out.print(d[i][j]+" ");
      System.out.println();
    }
    System.out.println("\n");
  }// afism(char[][]...)
}// class
```

Varianta 2:

```
import java.io.*; // problema reala ...
class Oracol
{
  static final char sus='|', stanga='-', diag='*';
  static int[][] a;
  static char[][] d;
  static String x,y;
  static boolean[] xx=new boolean[101];
```

```
  static boolean[] yy=new boolean[101];
  static int m,n,nc;

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    BufferedReader br=new BufferedReader(new FileReader("oracol.in"));
    PrintWriter out=new PrintWriter(
      new BufferedWriter( new FileWriter("oracol.out")));

    nc=Integer.parseInt(br.readLine());

    x=br.readLine().replace(" ",""); // elimina spatiile ... de la sfarsit !
    m=x.length();

    for(k=1;k<=nc;k++)
    {
      y=br.readLine().replaceAll(" ",""); // elimina spatiile ... daca sunt!
      n=y.length();

      matrad();
      for(i=1;i<=m;i++) xx[i]=false;
      for(j=1;j<=n;j++) yy[j]=false;
      osol(m,n);

      for(i=1;i<=m;i++) if(!xx[i]) out.print(i+" ");
      out.println();
      for(j=1;j<=n;j++) if(!yy[j]) out.print(j+" ");
      out.println();
    }
    out.close();
  }// main(...)

  static void matrad()
  {
    int i,j;
    a=new  int[m+1][n+1];
    d=new char[m+1][n+1];
    for(i=1;i<=m;i++)
    for(j=1;j<=n;j++)
    if(x.charAt(i-1)==y.charAt(j-1))
    {
      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;
    }
  }// matrad()

  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) xx[lin]=yy[col]=true;
  }// osol(...)

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }// max(...)
}// class
```

16.2.13 Pavări - ONI2001 clasa a X-a

prof. Doru Popescu Anastasiu, Slatina

Se dă un dreptunghi cu lungimea egală cu $2N$ centimetri şi lăţimea egală cu 3 centimetri .

Cerinţă

Să se determine numărul M al pavărilor distincte cu dale dreptunghiulare care au lungimea egală cu un centimetru şi lăţimea egală cu 2 centimetri.

Datele de intrare

Fişier de intrare: **pavari.in**

Linia 1: N - număr natural nenul, reprezentânnd jumătatea lungimii dreptunghiului.

Datele de ieşire

Fişier de ieşire: **pavari.out**

Linia 1: M - număr natural nenul, reprezentând numărul modalităţilor de a pava dreptunghiul.

Restricţii şi precizări

• $1 \leq N \leq 100$

Exemplu

pavari.in	pavari.out
2	11

Timp maxim de executare: 1 secundă/test

Codul sursă *

```
import java.io.*;   // x[n]=x[n-1]+2*y[n-1];  x[1]=3;
class Pavari        // y[n]=y[n-1]+x[n];      y[1]=4;
{
  public static void main(String[] args) throws IOException
  {
    int k,kk,n;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("pavari9.in")));
    st.nextToken(); n=(int)st.nval;
    int[] xv,xn,yv,yn;
    xv=new int[1];
    yv=new int[1];

    xv[0]=3;
    yv[0]=4;
    xn=xv;
    for(k=2;k<=n;k++)
    {
      xn=suma(xv,suma(yv,yv));
      yn=suma(yv,xn);
      xv=xn;
      yv=yn;
    }

    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("pavari.out")));
    for(kk=xn.length-1;kk>=0;kk--) out.print(xn[kk]);
    out.close();
  }// main(...)

  static int[] suma(int[] x, int[] y)
  {
    int nx=x.length,ny=y.length,i,t;
    int nz;
    if(nx>ny)  nz=nx+1; else nz=ny+1;
```

```
    int[] z=new int[nz];

    t=0;
    for(i=0;i<nz;i++)
    {
      z[i]=t;
      if(i<nx) z[i]+=x[i];
      if(i<ny) z[i]+=y[i];
      t=z[i]/10;
      z[i]=z[i]%10;
    }

    if(z[nz-1]!=0) return z;
    else
    {
      int[] zz=new int[nz-1];
      for(i=0;i<nz-1;i++) zz[i]=z[i];
      return zz;
    }
  }//suma
}
```

16.2.14 Balanţa ONI2002 clasa a X-a

Gigel are o "balanţă" mai ciudată pe care vrea să o echilibreze. De fapt, aparatul este diferit de orice balanţă pe care aţi vazut-o până acum.

Balanţa lui Gigel dispune de două braţe de greutate neglijabilă şi lungime 15 fiecare. Din loc în loc, la aceste braţe sunt ataşate cârlige, pe care Gigel poate atârna greutăţi distincte din colecţia sa de G greutăţi (numere naturale între 1 şi 25). Gigel poate atârna oricâte greutăţi de orice cârlig, dar trebuie să folosească toate greutăţile de care dispune.

Folosindu-se de experienţa participării la Olimpiada Naţională de Informatică, Gigel a reuşit să echilibreze balanţa relativ repede, dar acum doreşte să ştie în câte moduri poate fi ea echilibrată.

Cerinţă

Cunoscând amplasamentul cârligelor şi setul de greutăţi pe care Gigel îl are la dispoziţie, scrieţi un program care calculează în câte moduri se poate echilibra balanţa.

Se presupune că este posibil să se echilibreze balanţa (va fi posibil pe toate testele date la evaluare).

Datele de intrare

Fişierul de intrare **balanta.in** are următoarea structură:

• pe prima linie, numărul C de cârlige şi numărul G de greutăţi, valori separate prin spaţiu;

• pe următoarea linie, C numere întregi, distincte, separate prin spaţiu, cu valori cuprinse între -15 şi 15 inclusiv, reprezentând amplasamentele cârligelor faţă de centrul balanţei; valoarea absolută a numerelor reprezintă distanţa faţă de centrul balanţei, iar semnul precizează braţul balanţei la care este ataşat crligul, "-" pentru braţul stâng şi "+" pentru braţul drept;

• pe următoarea linie, G numere naturale distincte, cuprinse între 1 şi 25 inclusiv, reprezentând valorile greutăţilor pe care Gigel le va folosi pentru a echilibra balanţa.

Datele de ieşire

Fişierul de ieşire **balanta.out** conţine o singură linie, pe care se află un număr natural M, numărul de variante de plasare a greutăţilor care duc la echilibrarea balanţei.

Restricţii şi precizări

• $2 \leq C \leq 20$, $2 \leq G \leq 20$;

• greutăţile folosite au valori naturale între 1 şi 25;

• numărul M cerut este între 1 şi 100.000.000;

• celelalte restricţii (lungimea braţelor balanţei etc.) au fost prezentate anterior.

• balanţa se echilibrează dacă suma produselor dintre greutăţi şi coordonatele unde ele sunt plasate este 0 (suma momentelor greutăţilor faţă de centrul balanţei este 0).

Exemplu

balanta.in	balanta.out
2 4 -2 3 3 4 5 8	2

Timp maxim de executare: 1 secundă/test

Indicaţii de rezolvare *

Soluţia comisiei

Problema se rezolva prin metoda *programării dinamice*.

Se calculează în câte moduri se poate scrie fiecare sumă j, folosind primele i greutăţi. Iniţial $i = 0$ şi suma 0 se poate obţine într-un singur mod, restul sumelor în 0 moduri.

Urmeaza G paşi. La fiecare astfel de pas i se calculează în câte moduri putem obţine fiecare sumă introducând o nouă greutate - a i-a - în toate configuraţiile precedente. Practic, dacă suma S s-a obtinut cu primele $i-1$ greutăţi în M moduri, punând greutatea i pe cârligul k se va obţine suma $S+(greutate[i]*coordonata[k])$

în M moduri (la care, evident, se pot adauga alte moduri de obţinere plasând greutatea i pe un alt cârlig şi folosind suma respectivă).

În acest mod s-ar construi o matrice cu G linii şi $2 * (sumamaxima) + 1$ coloane, cu elemente numere întregi pe 32 de biţi; de fapt se memoreaza doar ultimele două linii (fiecare linie se obţine din precedenta). Suma maximă este $15 * 25 * 20 = 7500$, deci o linie se incadrează în mai puţin de $64K$.

Rezultatul final se obţine pe ultima linie, în coloana asociată sumei 0.

GInfo 12/6 octombrie 2002

Se observă că suma momentelor greutăţilor este cuprinsă între -6000 şi 6000 (dacă avem 20 de greutăţi cu valoarea 20 şi acestea sunt amplasate pe cel mai îndepărtat cârlig faţă de centrul balanţei, atunci modulul sumei momentelor forţelor este $152020 = 6000$). Ca urmare, putem păstra un şir a ale cărui valori a_i vor conţine numărul posibilităţilor ca suma momentelor greutăţilor să fie i.

Indicii şirului vor varia între -6000 şi 6000. Pentru a îmbunătăţi viteza de execuţie a programului, indicii vor varia între $-300g$ şi $300g$, unde g este numărul greutăţilor. Pot fi realizate îmbunătăţiri suplimentare dacă se determină distanţele maxime faţă de mijlocul balanţei ale celor mai îndepărtate cârlige de pe cele două talere şi suma totală a greutăţilor. Dacă distanţele sunt d_1 şi d_2, iar suma este s, atunci indicii vor varia între $-d_1 s$ şi $d_2 s$.

Iniţial, pe balanţă nu este agăţată nici o greutate, aşadar suma momentelor greutăţilor este 0. Ca urmare, iniţial valorile a_i vor fi 0 pentru orice indice nenul şi $a_0 = 1$ (există o posibilitate ca iniţial suma momentelor greutăilor să fie 0 şi nu există nici o posibilitate ca ea să fie diferită de 0).

În continuare, vom încerca să amplasăm greutăţile pe cârlige. Fiecare greutate poate fi amplasată pe oricare dintre cârlige. Să presupunem că la un moment dat există a_i posibilităţi de a obţine suma i. Dacă vom amplasa o greutate de valoare g pe un cârlig aflat la distanţa d faţă de centrul balanţei, suma momentelor greutăţilor va creşte sau va scădea cu gd (în funcţie de braţul pe care se află cârligul). Ca urmare, după amplasarea noii greutăţi există a_i posibilităţi de a obţine suma $i+gd$. Considerăm că şirul b va conţine valori care reprezintă numărul posibilităţilor de a obţine sume ale momentelor forţelor după amplasarea greutăţii curente. Înainte de a testa posibilităile de plasare a greutăţii, şirul b va conţine doar zerouri. Pentru fiecare pereche (i, d), valoarea b_{i+gd} va creşte cu a_i. După considerarea tuturor perechilor, vom putea trece la o nouă greutate.

Valorile din şirul b vor fi salvate în şirul a, iar şirul b va fi reiniţializat cu 0. Rezultatul final va fi dat de valoarea a_0 obţinută după considerarea tuturor greutăţilor disponibile.

Analiza complexităţii

Pentru studiul complexităţii vom nota numărul greutăţilor cu g, iar cel al cârligelor cu c.

Citirea datelor de intrare corespunzătoare cârligelor şi greutaţilor se realizează în timp liniar, deci ordinul de complexitate al acestor operaţii este $O(g)$, respectiv $O(c)$.

Singura mărime care nu este considerată constantă şi de care depinde numărul de posibilităţi de a obţine sumele este numărul greutăţilor. Aşadar, vom considera că ordinul de complexitate al traversării şirului a este $O(g)$.

Numărul de traversări este dat tot de numărul greutăţilor disponibile, deci vom avea $O(g)$ traversări.

În timpul traversării vom considera toate cârligele pentru fiecare element al şirului. Ca urmare, ordinul de complexitate al operaţiilor efectuate asupra unui element într-o parcurgere este $O(c)$. Rezultă că ordinul de complexitate al unei parcurgeri este $O(g)O(c) = O(gc)$, în timp ce ordinul de complexitate al întregii operaţii care duce la obţinerea rezultatului este $O(g)O(gc) = O(g2c)$.

Afişarea numărului de posibilităţi de a echilibra balanţa se realizează în timp constant.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(c) + O(g) + O(g^2c) + O(1) = O(g^2c)$.

Codul sursă

```
import java.io.*;
class Balanta
{
  static long[] a=new long[15001];   // a[i] i = -7500, 7500
  static long[] b=new long[15001];   // sir auxiliar (a+7500)!!!
  static int[] carlig=new int[20];   // coordonatele carligelor
  static int[] greutate=new int[20]; // valorile greutatilor
  static int nrCarlige, nrGreutati;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    citesteDatele();
    determinaSolutia();
    scrieSolutia();
    t2=System.currentTimeMillis();
    System.out.println("TIMP = "+(t2-t1)+" milisecunde");
  }// main()

  static void citesteDatele() throws IOException
  {
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("balanta9.in")));
    st.nextToken(); nrCarlige=(int)st.nval;
```

```
    st.nextToken(); nrGreutati=(int)st.nval;

    for(int i=0;i<nrCarlige;i++)
    {
      st.nextToken(); carlig[i]=(int)st.nval;
    }
    for(int i=0;i<nrGreutati;i++)
    {
      st.nextToken(); greutate[i]=(int)st.nval;
    }
  }// citesteDate()

  static void scrieSolutia() throws IOException
  {
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("balanta9.out")));
    out.print(a[7500+0]);
    out.close();
  }// scrieSolutia()

  static void determinaSolutia()
  {
    int i,j,k;
    a[7500+0]=1; // initial balanta este echilibrata
    for(i=0;i<nrGreutati;i++)
    {
      for(j=-7500;j<=7500;j++)
        if(a[7500+j]!=0)
          for(k=0;k<nrCarlige;k++)
            b[7500+j+carlig[k]*greutate[i]]+=a[7500+j];
      for (j=-7500;j<=7500;j++)
      {
        a[7500+j]=b[7500+j];
        b[7500+j]=0;
      }
    }
  }// determinaSolutia()
}// class
```

16.2.15 Aliniere ONI2002 clasa a X-a

În armată, o companie este alcătuită din n soldaţi. La inspecţia de dimineaţă soldaţii stau aliniaţi în linie dreaptă în faţa căpitanului. Acesta nu e mulţumit de

ceea ce vede; e drept că soldaţii sunt aşezaţi în ordinea numerelor de cod $1, 2, ..., n$ din registru, dar nu în ordinea înălţimii. Căpitanul cere câtorva soldaţi să iasă din rând, astfel ca cei rămaşi, fără a-şi schimba locurile, doar apropiindu-se unul de altul (pentru a nu rămâne spaţii mari între ei) să formeze un şir în care fiecare soldat vede privind de-a lungul şirului, cel puţin una din extremităţi (stânga sau dreapta). Un soldat vede o extremitate dacă între el şi capătul respectiv nu există un alt soldat cu înălţimea mai mare sau egală ca a lui.

Cerinţă

Scrieţi un program care determină, cunoscând înălţimea fiecărui soldat, numărul minim de soldaţi care trebuie să părăsească formaţia astfel ca şirul rămas să îndeplinească condiţia din enunţ.

Datele de intrare

Pe prima linie a fişierului de intrare **aliniere.in** este scris numărul n al soldaţilor din şir, iar pe linia următoare un şir de n numere reale, cu maximum 5 zecimale fiecare şi separate prin spaţii. Al k-lea număr de pe această linie reprezintă înălţimea soldatului cu codul k ($1 \leq k \leq n$).

Datele de ieşire

Fişierul **aliniere.out** va conţine pe prima linie numărul soldaţilor care trebuie să părăsească formaţia, iar pe linia următoare codurile acestora în ordine crescătoare, separate două câte două printr-un spaţiu. Dacă există mai multe soluţii posibile, se va scrie una singură.

Restricţii şi precizări

- $2 \leq n \leq 1000$
- înălţimile sunt numere reale în intervalul $[0.5; 2.5]$.

Exemplu

aliniere.in	**aliniere.out**
8 1.86 1.86 1.30621 2 1.4 1 1.97 2.2	4 1 3 7 8

Explicaţie

Rămân soldaţii cu codurile 2, 4, 5, 6 având înălţimile 1.86, 2, 1.4 şi 1.

Soldatul cu codul 2 vede extremitatea stângă.

Soldatul cu codul 4 vede ambele extremităţi.

Soldaţii cu codurile 5 şi 6 văd extremitatea dreaptă.

Timp maxim de executare: 1 secundă/test

Indicaţii de rezolvare

Soluţia comisiei

Problema se rezolvă prin metoda *programării dinamice*.

Se calculează, pentru fiecare element, lungimea celui mai lung subşir strict crescător care se termină cu el şi lungimea celui mai lung subşir strict descrescător care începe cu el.

Solutia constă în păstrarea a două astfel de subşiruri de soldaţi (unul crescător şi unul descrescător) pentru DOI soldaţi de aceeaşi înălţime (eventual identici) şi eliminarea celorlalţi. Soldaţii din primul subşir privesc spre stanga, ceilalţi spre dreapta. Primul subşir se termină inainte de a începe al doilea. Se au in vedere cazurile particulare.

Deoarece s-a considerat că o parte din concurenţi vor rezolva problema pentru un singur soldat central (toţi ceilalti soldaţi păstrati având înaltimea mai mică) şi nu vor observa cazul în care se pot păstra doi soldaţi de aceeaşi înalţime, majoritatea testelor se încadrează în acest caz.

GInfo 12/6 octombrie 2002

Pentru fiecare soldat vom determina cel mai lung subşir strict crescător (din punct de vedere al înălţimii) de soldaţi care se termină cu el, respectiv cel mai lung subşir strict descrescător de soldaţi care urmează după el.

După această operaţie, vom determina soldatul pentru care suma lungimilor celor două şiruri este maximă. Chiar dacă s-ar părea că în acest mod am găsit soluţia problemei, mai există o posibilitate de a mări numărul soldaţilor care rămân în şir. Să considerăm soldatul cel mai înalt în şirul rămas (cel căruia îi corespunde suma maximă). Acesta poate privi fie spre stânga, fie spre dreapta şirului. Din aceste motive, la stânga sau la dreapta sa poate să se afle un soldat de aceeaşi înălţime; unul dintre cei doi va privi spre dreapta, iar celălalt spre stânga. Totuşi, nu putem alege orice soldat cu aceeaşi înălţime, ci doar unul pentru care lungimea şirului strict crescător (dacă se află spre stânga) sau a celui strict descrescător (dacă se află spre dreapta) este aceeaşi cu lungimea corespunzătoare şirului strict crescător, respectiv strict descrescător, corespunzătoare celui mai înalt soldat dintre cei rămaşi în şir.

După identificarea celor doi soldaţi de înălţimi egale (sau demonstrarea faptului că nu există o pereche de acest gen care să respecte condiţiile date) se marchează toţi soldaţii din cele două subşiruri. Ceilalţi soldaţi vor trebui să părăsească formaţia.

Analiza complexităţii

Citirea datelor de intrare se realizează în timp liniar, deci ordinul de complexitate al acestei operaţii este $O(n)$.

Chiar dacă exist algoritmi eficienţi (care rulează n timp liniar-logaritmic) de determinare a celui mai lung subşir ordonat, timpul de execuţie admis ne permite folosirea unui algoritm simplu, cu ordinul de complexitate $O(n^2)$. Acesta va fi aplicat de două ori, după care se va căuta valoarea maximă a sumei lungimilor şirurilor corespunzătoare unui soldat; aşadar identificarea soldatului care poate privi în ambele direcţii este o operaţie cu ordinul de complexitate $O(n^2)+O(n^2)+O(n) = O(n^2)$.

Urmează eventuala identificare a unui alt soldat de aceeaşi înălţime care respectă condiţiile referitioare la lungimile subşirurilor. Pentru aceasta se parcurge

şirul soldaţilor de la soldatul identificat anterior spre extremităţi; operaţia necesită un timp liniar.

Deteminarea celor două subşiruri se realizează în timp liniar dacă, în momentul construirii celor două subşiruri, se păstrează predecesorul, respectiv succesorul fiecărui soldat. În timpul parcurgerii subşirurilor sunt marcaţi soldaţii care rămân în formaţie.

Pentru scrierea datelor de ieşire se parcurge şirul marcajelor şi sunt identificaţi soldaţii care părăsesc formaţia. Ordinul de complexitate al acestei operaţii este $O(n)$.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(n) + O(n^2) + O(n) + O(n) + O(n) = O(n^2)$.

Codul sursă

```
import java.io.*;
class Aliniere
{
  static final String fisi="aliniere.in";
  static float[] inalt;     // inaltimile soldatilor
  static int ns, nsr;       // nr soldati, nr soldati ramasi
  static int[] predCresc;   // predecesor in subsirul crescator
  static int[] lgCresc;     // lungimea sirului crescator care se termina cu i
  static int[] succDesc;    // succesor in subsirul descrescator
  static int[] lgDesc;      // lungimea sirului descrescator care urmeaza dupa i
  static boolean[] ramas;   // ramas in sir

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    citescDate();
    subsirCresc();
    subsirDesc();
    alegeSoldati();
    scrieRezultate();
    t2=System.currentTimeMillis();
    System.out.println("TIME = "+(t2-t1)+" millisec ");
  }// main()

  static void citescDate() throws IOException
  {
    StreamTokenizer st=new StreamTokenizer(
```

```
                    new BufferedReader(new FileReader(fisi)));
  st.nextToken(); ns=(int)st.nval;

  predCresc=new int[ns];
  lgCresc=new int[ns];
  succDesc=new int[ns];
  lgDesc=new int[ns];
  ramas=new boolean[ns];
  inalt=new float[ns];

  for(int i=0;i<ns;i++) {st.nextToken(); inalt[i]=(float)st.nval;}
}//citescDate()

static void subsirCresc()
{
  int i,j;
  lgCresc[0]=1;
  predCresc[0]=-1;
  for(i=1;i<ns;i++)
  {
    lgCresc[i]=1;           // subsirul formar doar din i
    predCresc[i]=-1;        // nu are predecesor
    for (int j=0;j<i;j++)
    if(inalt[j]<inalt[i])
      if(lgCresc[i]<lgCresc[j]+1) // sir mai lung
      {
        lgCresc[i]=lgCresc[j]+1;
        predCresc[i] = j;
      }
  }
}//subsirCresc()

static void subsirDesc()
{
  int i,j;
  lgDesc[ns-1]=0;        // nu exista nici un soldat mai mic dupa ns-1
  succDesc[ns-1]=-1;     // ns-1 nu are succesor
  for(i=ns-2;i>=0;i--)
  {
    lgDesc[i]=0;        // nu exista nici un soldat mai mic dupa i
    succDesc[i]=-1;     // i nu are succesor
    for(j=ns-1;j>i;j--)
      if(inalt[j]<inalt[i])         // soldat mai mic
        if(lgDesc[i]<lgDesc[j]+1)   // sir mai lung
```

```
        {
          lgDesc[i]=lgDesc[j]+1;    // actualizarea lg subsir
          succDesc[i]=j;            // actualizare succesor
        }
    }
  }// subsirDesc()

  static void alegeSoldati()
  {
    int i;
    // este posibil ca in mijloc sa fie doi soldati cu inaltimi egale
    int im=-1;        // indicele soldatului din mijloc
    int ic, id;       // indicii care delimiteaza in interior cele doua subsiruri
    for(i=0;i<ns;i++)
    if(lgCresc[i]+lgDesc[i]>nsr)
    {
      nsr=lgCresc[i]+lgDesc[i];
      im=i;
    }

    // in "mijlocul" sirului se pot afla doi soldati cu aceeasi inaltime
    ic=im;
    id=im;

    // caut in stanga un subsir cu aceeasi lungime --> soldat cu aceeasi inaltime
    for(i=im-1;i>=0;i--)
    if(lgCresc[ic]==lgCresc[i]) ic=i;

    // caut in dreapta un subsir cu aceeasi lungime --> soldat cu aceeasi inaltime
    for(i=im+1;i<ns;i++)
    if(lgDesc[id]==lgDesc[i]) id=i;
    if(ic!=id)          // in "mijloc" sunt doi soldati cu aceeasi inaltime
    nsr++;
    while(id!=-1)       // indice descrescator
    {
      ramas[id]=true;
      id=succDesc[id];
    }
    while(ic!=-1)      // indice crescator
    {
      ramas[ic] = true;
      ic=predCresc[ic];
    }
  }// alegeSoldati()
```

```
  static void scrieRezultate() throws IOException
  {
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("aliniere.out")));
    out.println(ns- nsr);
    for(int i=0;i<ns;i++) if(!ramas[i]) out.print((i+1)+" ");
    out.close();
  }// scriuRezultate()
}// class
```

16.2.16 Munte - ONI2003 cls 10

Într-o zonă montană se doreşte deschiderea unui lanţ de telecabine. Staţiile de telecabine pot fi înfiinţate pe oricare din cele N vârfuri ale zonei montane. Vârfurile sunt date în ordine de la stânga la dreapta şi numerotate de la 1 la N, fiecare vârf i fiind precizat prin coordonata $X[i]$ pe axa OX şi prin înălţimea $H[i]$.

Se vor înfiinţa exact K staţii de telecabine. Staţia de telecabine i ($2 \leq i \leq K$) va fi conectată cu staţiile $i-1$ şi $i+1$; staţia 1 va fi conectată doar cu staţia 2, iar staţia K, doar cu staţia $K-1$. Staţia 1 va fi obligatoriu amplasată în vârful 1, iar staţia K în vârful N.

Se doreşte ca lanţul de telecabine să asigure legătura între vârful 1 şi vârful N. Mai mult, se doreşte ca lungimea totală a cablurilor folosite pentru conectare să fie minimă. Lungimea cablului folosit pentru a conecta două staţii este egală cu distanţa dintre ele. În plus, un cablu care uneşte două staţii consecutive nu poate avea lungimea mai mare decât o lungime fixată L.

O restricţie suplimentarăeste introdusă de formele de relief. Astfel, vârfurile i şi j ($i < j$) nu pot fi conectate direct dacă există un vârf v ($i < v < j$) astfel încât segmentul de dreapta care ar uni vâfurile i şi j nu ar trece pe deasupra vârfului v. În cazul în care cele trei vârfuri sunt coliniare, se consideră toate trei ca fiind staţii, chiar dacă distan c dintre vrfurile i şi j este mai mică decât L.

Cerinţă

Dându-se amplasarea celor N vrfuri ale lanţului muntos, stabiliţi o modalitate de dispunere a celor K staţii de telecabine astfel încât lungimea totală a cablurilor folosite pentru conectare să fie minimă, cu restricţiile de mai sus.

Se garantează că, pe toate testele date la evaluare, conectarea va fi posibilă.

Date de intrare

Prima linie a fişierului de intrare **munte.in** conţine trei numere întregi N, K şi L, separate prin spaţii, cu semnificaţiile de mai sus. Următoarele N linii conţin coordonatele vârfurilor; linia $i+1$ conţine coordonatele vârfului i, $X[i]$ şi $H[i]$, separate printr-un spaţiu.

Date de ieşire

În fişierul **munte.out** veţi afişa:

- pe prima linie lungimea totală minimă a cablurilor, rotunjită la cel mai apropiat numar întreg (pentru orice întreg Q, $Q.5$ se rotunjeste la $Q+1$);

- pe a doua linie K numere distincte între 1 şi N, ordonate crescător, numerele vârfurilor în care se vor înfiinţa staţii de telecabine. Dacă există mai multe variante, afişaţi una oarecare.

Restricţii şi precizări

$2 \leq N \leq 100$

$2 \leq K \leq 30$ şi $K \leq N$

$0 \leq L$, $X[i], H[i] \leq 100.000$ şi $X[i] < X[i+1]$

Exemplu

munte.in	**munte.out**
7 5 11	22
0 16	1 3 5 6 7
4 3	
6 8	
7 4	
12 16	
13 16	
14 16	

Explicaţii

- trasarea unui cablu direct între vârfurile 1 şi 5 ar fi contravenit restricţiei referitoare la lungimea maximă a unui cablu; în plus, s-ar fi obţinut o soluţie cu 2 staţii de telecabine în loc de 3 (deci soluţia ar fi invalidă şi pentru valori mari ale lui L);

- pentru a ilustra restricţia introdusă de formele de relief, precizăm că vârfurile 1 şi 4 nu au putut fi conectate direct datorită înălţimii vârfului 3. De asemenea, vârfurile 5 şi 7 nu au putut fi conectate direct datorită înălţimii vârfului 6.

Timp maxim de executare: 1 secundă/test.

Indicaţii de rezolvare - descriere soluţie *

Mihai Stroe, GInfo nr. 13/6 - octombrie 2003

Problema se rezolvă prin *metoda programării dinamice*.

Practic, problema se poate împărţi în două subprobleme. Primul pas constă în determinarea perechilor de vârfuri care pot fi unite printr-un cablu. Acest pas se rezolvă folosind cunoştinţe elementare de geometrie plană (ecuaţia dreptei, $y = a*x+b$). Se va obţine o matrice OK, unde $OK_{i,j}$ are valoarea 1 dacă vârfurile i şi j pot fi unite şi 0 în caz contrar. Folosind această matrice, se vor conecta vârfurile 1 şi N cu un lanţ de K staţii, astfel încât oricare două staţii consecutive să aibă OK-ul corespunzător egal cu 1.

Această subproblema se rezolvă prin *metoda programării dinamice* după cum urmează: se construieşte o matrice A cu K linii şi N coloane, unde $A_{i,j}$ reprezintă lungimea totală minimă a unui lanţ cu i staţii care conectează vârfurile 1 şi j.

Iniţial $A_{1,1} = 0$, $A_{1,i} = +\infty$ şi $A_{i,1} = +\infty$ pentru $i > 1$.

Pentru i cuprins între 2 şi N, componentele matricei se calculează astfel:

$$A_{i,j} = \min\{A_{i-1,v} + dist(v,j)\}, \text{ unde } v < j \text{ şi } OK_{v,j} = 1$$

Concomitent cu calculul elementelor $A_{i,j}$ se construieşte o matrice T, unde $T_{i,j}$ reprezintă v-ul care minimizează expresia de mai sus. Matricea T este folosită pentru reconstituirea soluţiei.

Lungimea totală minimă este regasită în $A_{K,N}$.

Subproblemele puteau fi tratate şi simultan, ceea ce complica implementarea.

Analiza complexităţii

Operaţia de citire a datelor are ordinul de complexitate $O(N)$.

Calculul matricei OK are ordinul de complexitate $O(N^3)$.

Algoritmul bazat pe *metoda programării dinamice* are ordinul de complexitate $O(N^2 \cdot K)$.

Reconstituirea soluţiei şi afişarea au ordinul de complexitate $O(N)$.

Deoarece $K \leq N$, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este $O(N^3)$.

Codul sursă

Prima variantă:

```
import java.io.*;        // cu mesaje pt depanare dar ... fara traseu
class Munte1
{
  static final int oo=Integer.MAX_VALUE;
  static int n,m,L;     // m=nr statii (in loc de K din enunt)

  static int[] x,h;
  static double[][] a;
  static int[][] t;
  static boolean[][] ok;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i;
    BufferedReader br=new BufferedReader(new FileReader("munte.in"));
    StreamTokenizer st=new StreamTokenizer(br);

    st.nextToken(); n=(int)st.nval;
```

```
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); L=(int)st.nval;

    x=new int[n+1];
    h=new int[n+1];
    ok=new boolean[n+1][n+1];      // implicit este false
    a=new double[m+1][n+1];
    t=new int[m+1][n+1];

    for(i=1;i<=n;i++)
    {
      st.nextToken(); x[i]=(int)st.nval;
      st.nextToken(); h[i]=(int)st.nval;
    }

    matriceaOK();
    afism(ok);

    matriceaA();

    afisSolutia();

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()

  static void matriceaA()
  {
    int i,j,k,kmin;
    double d,dkj,min;
    a[1][1]=0;
    for(i=2;i<=m;i++) a[i][1]=oo;
    for(j=2;j<=n;j++) a[1][j]=oo;
    afism(a);

    for(i=2;i<=m;i++)
    {
      for(j=2;j<=n;j++)
      {
        min=oo;
        kmin=-1;
        for(k=1;k<j;k++)
        {
          System.out.println(i+" "+j+" "+k+" "+ok[k][j]);
```

```
          if(ok[k][j])
          {
            dkj=dist(k,j);
            d=a[i-1][k]+dkj;
            System.out.println(i+" "+j+" "+k+" dkj="+dkj+" d="+d+" min="+min);
            if(d<min) { min=d;kmin=k; }
          }
        }// for k
        a[i][j]=min;
      }// for j
      System.out.println("Linia: "+i);
      afism(a);
    }// for i
  }// matriceaA()

  static double dist(int i, int j)
  {
    double d;
    d=(double)(x[i]-x[j])*(x[i]-x[j])+(double)(h[i]-h[j])*(h[i]-h[j]);
    return Math.sqrt(d);
  }// dist(...)

  static void matriceaOK()
  {
    int i,j,ij,x1,y1,x2,y2,sp1,sp2;
    for(i=1;i<=n-1;i++)
    {
      x1=x[i]; y1=h[i];
      for(j=i+1;j<=n;j++)
      {
        x2=x[j]; y2=h[j];
        ok[i][j]=ok[j][i]=true;
        for(ij=i+1;ij<=j-1;ij++)                    // i .. ij .. j
        {
          sp1=(0 -y1)*(x2-x1)-(y2-y1)*(x[ij]-x1);
          sp2=(h[ij]-y1)*(x2-x1)-(y2-y1)*(x[ij]-x1);
          if(sp1*sp2<=0)
          {
            ok[i][j]=ok[j][i]=false;
            System.out.println(i+" "+j+" ("+ij+")\t"+sp1+"\t"+sp2);
            break;
          }
          if(!ok[i][j]) break;
        }//for ij
```

```
        if(ok[i][j]) if(dist(i,j)>L+0.0000001) ok[i][j]=ok[j][i]=false;
      }//for j
    }// for i
  }// matriceaOK()

  static void afisSolutia() throws IOException
  {
    int i,j;
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter("munte.out")));
    out.println(a[m][n]);
    out.close();
  }//afisSolutia()

  static void afism(int[][] a)
  {
    int i,j;
    for(i=0;i<a.length;i++)
    {
      for(j=0;j<a[i].length;j++) System.out.print(a[i][j]+" ");
      System.out.println();
    }
    System.out.println();
  }// afism(...)

  static void afism(boolean[][] a)
  {
    int i,j;
    for(i=1;i<a.length;i++)
    {
      for(j=1;j<a[i].length;j++) System.out.print(a[i][j]+" ");
      System.out.println();
    }
    System.out.println();
  }// afism(...)

  static void afism(double[][] a)
  {
    int i,j;
    for(i=1;i<a.length;i++)
    {
      for(j=1;j<a[i].length;j++) System.out.print(a[i][j]+" ");
      System.out.println();
    }
```

```
    System.out.println();
  }// afism(...)
}// class
```

A doua variantă:

```
import java.io.*;   // FINAL: fara mesaje si cu traseu ... dar
class Munte2        // test 8 : P1(99,59) P2(171,96) P3(239,81) P4(300,78)
{           // solutia: 1 4 5 ... este gresita (1 4 nu trece de P2 si P3!)
  static final int oo=Integer.MAX_VALUE;
  static int n,m,L;              // m=nr statii (in loc de K din enunt)

  static int[] x,h;
  static double[][] a;
  static int[][] t;
  static boolean[][] ok;
  static int[] statia;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j;
    BufferedReader br=new BufferedReader(new FileReader("munte.in"));
    StreamTokenizer st=new StreamTokenizer(br);

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); L=(int)st.nval;

    x=new int[n+1];
    h=new int[n+1];
    ok=new boolean[n+1][n+1];
    a=new double[m+1][n+1];
    t=new int[m+1][n+1];
    statia=new int[m+1];
    for(i=1;i<=n;i++)
    {
      st.nextToken(); x[i]=(int)st.nval;
      st.nextToken(); h[i]=(int)st.nval;
    }

    matriceaOK();
```

```
    matriceaA();
    j=n;
    for(i=m;i>=1;i--) { statia[i]=j; j=t[i][j]; }
    afisSolutia();

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()

  static void matriceaA()
  {
    int i,j,k,kmin;
    double d,dkj,min;
    a[1][1]=0;
    for(i=2;i<=m;i++) a[i][1]=oo;
    for(j=2;j<=n;j++) a[1][j]=oo;

    for(i=2;i<=m;i++)
    {
      for(j=2;j<=n;j++)
      {
        min=oo;
        kmin=0;
        for(k=1;k<j;k++)
        {
          if(ok[k][j])
          {
            dkj=dist(k,j);
            d=a[i-1][k]+dkj;
            if(d<min) { min=d;kmin=k; }
          }
        }// for k
        a[i][j]=min;
        t[i][j]=kmin;
      }// for j
    }// for i
  }// matriceaA()

  static double dist(int i, int j)
  {
    double d;
    d=(double)(x[i]-x[j])*(x[i]-x[j])+(double)(h[i]-h[j])*(h[i]-h[j]);
    return Math.sqrt(d);
  }// dist(...)
```

```
  static void matriceaOK()
  {
    int i,j,ij,x1,y1,x2,y2;
    long sp1,sp2;       // 100.000*100.000=10.000.000.000 depaseste int !!!
    for(i=1;i<=n-1;i++)
    {
      x1=x[i]; y1=h[i];
      for(j=i+1;j<=n;j++)
      {
        x2=x[j]; y2=h[j];
        ok[i][j]=ok[j][i]=true;
        for(ij=i+1;ij<=j-1;ij++)        // i .. ij .. j
        {
          sp1=(0 -y1)*(x2-x1)-(y2-y1)*(x[ij]-x1);
          sp2=(h[ij]-y1)*(x2-x1)-(y2-y1)*(x[ij]-x1);
          if(sp1*sp2<=0)
          {
            ok[i][j]=ok[j][i]=false;
            break;
          }
          if(!ok[i][j]) break;
        }//for ij
       if(ok[i][j]) if(dist(i,j)>L) ok[i][j]=ok[j][i]=false;
      }//for j
    }// for i
  }// matriceaOK()

  static void afisSolutia() throws IOException
  {
    int i;
    PrintWriter out = new PrintWriter(
      new BufferedWriter(new FileWriter("munte.out")));
    out.println((int)(a[m][n]+0.5));
    for(i=1;i<=m;i++) out.print(statia[i]+" ");
    out.println();
    out.close();
  }//afisSolutia()
}// class
```

16.2.17 Lăcusta - OJI2005 clasa a X-a

Se consideră o matrice dreptunghiulară cu m linii şi n coloane, cu valori naturale. Traversăm matricea pornind de la colţul stânga-sus la colţul dreapta-jos.

O traversare constă din mai multe deplasări. La fiecare deplasare se execută un salt pe orizontală şi un pas pe verticală. Un salt înseamnă că putem trece de la o celulă la oricare alta aflată pe aceeaşi linie, iar un pas înseamnă că putem trece de la o celulă la celula aflată imediat sub ea. Excepţie face ultima deplasare (cea în care ne aflăm pe ultima linie), când vom face doar un salt pentru a ajunge în colţul dreapta-jos, dar nu vom mai face şi pasul corespunzător. Astfel traversarea va consta din vizitarea a $2m$ celule.

Cerinţă

Scrieţi un program care să determine suma minimă care se poate obţine pentru o astfel de traversare.

Datele de intrare

Fişierul de intrare **lacusta.in** conţine pe prima linie două numere naturale separate printr-un spaţiu m n, reprezentând numărul de linii şi respectiv numărul de coloane ale matricei. Pe următoarele m linii este descrisă matricea, câte n numere pe fiecare linie, separate prin câte un spaţiu.

Datele de ieşire

Fişierul de ieşire **lacusta.out** va conţine o singură linie pe care va fi scrisă suma minimă găsită.

Restricţii şi precizări

- $1 \leq m, n \leq 100$
- Valorile elementelor matricei sunt numere întregi din intervalul $[1, 255]$.

Exemple

lacusta.in	lacusta.out	Explicatie
4 5 3 4 5 7 9 6 6 3 4 4 6 3 3 9 6 6 5 3 8 2	28	Drumul este: $(1,1) \rightarrow (1,3) \rightarrow$ $(2,3) \rightarrow (2,2) \rightarrow$ $(3,2) \rightarrow (3,3) \rightarrow$ $(4,3) \rightarrow (4,5)$

Timp maxim de executare: 1 secundă/test

Indicaţii de rezolvare *

Ginfo nr. 15/3 martie 2005

Pentru rezolvarea acestei probleme vom utiliza metoda *programării dinamice*.

Vom nota prin A matricea dată şi vom construi o matrice B ale cărei elemente b_{ij} vor conţine sumele minime necesare pentru a ajunge în celula (i, j) pornind din celula $(i-1, j)$.

Vom completa iniţial elementele de pe a doua linie a matricei B. Valoarea $b_{2,1}$ va fi ∞ deoarece în această celulă nu se poate ajunge. Valorile celorlalte elemente b_{2i} vor fi calculate pe baza formulei: $b_{2,i} = a_{1,1} + a_{1,i} + a_{2,i}$.

Pentru celelalte linii, valorile b_{ij} vor fi calculate pe baza formulei:

$$b_{i,j} = a_{i,j} + a_{i-1,j} + min(b_{i-1,k}),$$

unde k variază între 1 şi n. Evident, relaţia nu este valabilă pentru elementul de pe coloana k care corespunde minimului, deoarece nu se poate coborî direct, ci trebuie efectuat un salt orizontal. În această situaţie vom alege al doilea minim de pe linia anterioară.

În final alegem minimul valorilor de pe ultima linie a matricei B (fără a lua în considerare elementul de pe ultima coloană a acestei linii) la care adaugăm valoarea a_{mn}.

Coduri sursă *

Prima variantă:

```
import java.io.*;
class Lacusta1
{
  static final int oo=100000;
  static int m,n;
  static int[][] a,b; // 0 <= i <= m-1;  0 <= j <= n-1

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j;

    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("lacusta.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("lacusta.out")));

    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;

    a=new int[m][n];
    b=new int[m][n];

    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
      {
```

```
        st.nextToken(); a[i][j]=(int)st.nval;
      }
    for(i=0;i<m;i++) for(j=0;j<n;j++) b[i][j]=oo;

    // prima linie (i=0) din b este oo
    // a doua linie (i=1) din b
    for(j=1;j<n;j++) b[1][j]=a[0][0]+a[0][j]+a[1][j];

    // urmatoarele linii din b
    for(i=2;i<m;i++)
      for(j=0;j<n;j++)
        b[i][j]=a[i][j]+a[i-1][j]+minLinia(i-1,j);

    // "obligatoriu" (!) si ultima linie (i=n-1) dar ... fara coborare
    b[m-1][n-1]=minLinia(m-1,n-1)+a[m-1][n-1];

    out.println(b[m-1][n-1]);
    out.close();
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()

  static int minLinia(int ii, int jj) // min pe linia=ii fara pozitia jj==col
  {
    int j,min=oo;
    for(j=0;j<n;j++)
      if(j!=jj)
        if(b[ii][j]<min) min=b[ii][j];
    return min;
  }// minLinia(...)
}// class
```

A doua variantă:

```
import java.io.*; // suplimentar ... si traseul !
class Lacusta2
{
  static final int oo=100000;
  static int m,n;
  static int[][] a,b; // 1 <= i <= m;  1 <= j <= n

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
```

```
t1=System.currentTimeMillis();

int i,j,min,jmin,j0;

StreamTokenizer st=new StreamTokenizer(
  new BufferedReader(new FileReader("lacusta.in")));
PrintWriter out=new PrintWriter(
  new BufferedWriter(new FileWriter("lacusta.out")));
st.nextToken(); m=(int)st.nval;
st.nextToken(); n=(int)st.nval;
a=new int[m+1][n+1];
b=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;
}
for(i=1;i<=m;i++) for(j=1;j<=n;j++) b[i][j]=oo;

// prima linie (i=1) din b este oo
// a doua linie (i=2) din b
for(j=2;j<=n;j++) b[2][j]=a[1][1]+a[1][j]+a[2][j];

// urmatoarele linii din b
for(i=3;i<=m;i++)
for(j=1;j<=n;j++) b[i][j]=a[i][j]+a[i-1][j]+minLinia(i-1,j);

// "obligatoriu" (!) si ultima linie (i=n) dar ... fara coborare
b[m][n]=minLinia(m,n)+a[m][n];

out.println(b[m][n]);
out.close();

jmin=-1; // initializare aiurea !
j0=1; // pentru linia 2
System.out.print(1+" "+1+" --> ");
for(i=2;i<=m-1;i++) // liniile 2 .. m-1
{
  min=oo;
  for(j=1;j<=n;j++)
  if(j!=j0)
  if(b[i][j]<min) { min=b[i][j]; jmin=j;}
  System.out.print((i-1)+" "+jmin+" --> ");
```

```
      System.out.print(i+" "+jmin+" --> ");
      j0=jmin;
    }

    j0=n;
    min=oo;
    for(j=1;j<n;j++)
    if(j!=j0)
    if(b[i][j]<min) { min=b[i][j]; jmin=j;}
    System.out.print((i-1)+" "+jmin+" --> ");
    System.out.print(i+" "+jmin+" --> ");
    System.out.println(m+" "+n);

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()

  static int minLinia(int ii, int jj) // min pe linia=ii fara pozitia jj==col
  {
    int j,min=oo;
    for(j=1;j<=n;j++)
      if(j!=jj)
        if(b[ii][j]<min) min=b[ii][j];
    return min;
  }// minLinia(...)
}// class
```

Varianta 3:

```
import java.io.*; // fara matricea de costuri (economie de "spatiu")
class Lacusta3    // traseul este ... pentru depanare
{                 // daca se cere ... se poate inregistra si apoi ...
  static final int oo=100000;
  static int m,n;
  static int[][] a; // 1 <= i <= m;  1 <= j <= n

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j;
    int minc,minsol,jmin,j0;
```

```
StreamTokenizer st=new StreamTokenizer(
  new BufferedReader(new FileReader("lacusta.in")));
PrintWriter out=new PrintWriter(
  new BufferedWriter(new FileWriter("lacusta.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;
}

minsol=oo;
System.out.print(1+" "+1+" --> ");

// a doua linie (i=2)
minc=oo;
jmin=-1;
for(j=2;j<=n;j++)
  if(a[1][1]+a[1][j]+a[2][j]<minc) {minc=a[1][1]+a[1][j]+a[2][j]; jmin=j;}
System.out.print(1+" "+jmin+" --> ");
System.out.print(2+" "+jmin+" --> ");
minsol=minc;
j0=jmin;

jmin=-1; // initializare aiurea !
for(i=3;i<=m-1;i++)
{
  minc=oo;
  for(j=1;j<=n;j++)
  if(j!=j0)
    if(a[i-1][j]+a[i][j]<minc)
      {minc=a[i-1][j]+a[i][j]; jmin=j;}
  System.out.print((i-1)+" "+jmin+" --> ");
  System.out.print(i+" "+jmin+" --> ");
  minsol+=minc;
  j0=jmin;
}

j0=n;
minc=oo;
for(j=1;j<=n;j++)
```

```
      if(j!=j0)
        if(a[m-1][j]+a[m][j]<minc)
          {minc=a[m-1][j]+a[m][j]; jmin=j;}
    System.out.print((m-1)+" "+jmin+" --> ");
    System.out.print(m+" "+jmin+" --> ");
    minsol+=minc+a[m][n];
    System.out.println(m+" "+n);

    out.println(minsol);
    out.close();

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()
}// class
```

Varianta 4:

```
import java.io.*; // fara matricea de costuri (economie de "spatiu")
class Lacusta4    // si ... fara matricea initiala (numai doua linii din ea !)
{                 // calculez pe masura ce citesc cate o linie !
  static final int oo=100000;
  static int m,n;
  static int[][] a; // 1 <= i <= m;  1 <= j <= n

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j,ii,jj;
    int minc,minsol,jmin,j0;

    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("lacusta.in")));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("lacusta.out")));
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;
    a=new int[2][n+1];

    for(i=1;i<=2;i++) // citesc numai primele doua linii
    for(j=1;j<=n;j++)
    {
```

```
  st.nextToken(); a[i%2][j]=(int)st.nval;
}

minsol=oo;
System.out.print(1+" "+1+" --> ");

// a doua linie (i=2)
minc=oo;
jmin=-1;
for(j=2;j<=n;j++)
  if(a[1%2][1]+a[1][j]+a[2%2][j]<minc)
    {minc=a[1%2][1]+a[1%2][j]+a[2%2][j]; jmin=j;}
System.out.print(1+" "+jmin+" --> ");
System.out.print(2+" "+jmin+" --> ");
minsol=minc;
j0=jmin;

jmin=-1;              // initializare aiurea !
for(i=3;i<=m-1;i++)  // citesc mai departe cate o linie
{
  for(j=1;j<=n;j++) { st.nextToken(); a[i%2][j]=(int)st.nval; }
  minc=oo;
  for(j=1;j<=n;j++)
    if(j!=j0)
      if(a[(i-1+2)%2][j]+a[i%2][j]<minc)
        {minc=a[(i-1+2)%2][j]+a[i%2][j]; jmin=j;}
  System.out.print((i-1)+" "+jmin+" --> ");
  System.out.print(i+" "+jmin+" --> ");
  minsol+=minc;
  j0=jmin;
}

//citesc linia m
for(j=1;j<=n;j++) { st.nextToken(); a[m%2][j]=(int)st.nval; }

j0=n;
minc=oo;
for(j=1;j<=n;j++)
  if(j!=j0)
    if(a[(m-1+2)%2][j]+a[m%2][j]<minc)
      {minc=a[(i-1+2)%2][j]+a[i%2][j]; jmin=j;}
System.out.print((i-1)+" "+jmin+" --> ");
System.out.print(i+" "+jmin+" --> ");
minsol+=minc+a[m%2][n];
```

```
    System.out.println(m+" "+n);

    out.println(minsol);
    out.close();

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()
}// class
```

Varianta 5:

```
import java.io.*; // numai o linie din matricea initiala !
class Lacusta5
{
  static final int oo=100000;
  static int m,n;
  static int[] a; // 1 <= j <= n

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j,ii,jj,a11,aa;
    int minc,minsol,jmin,j0;

    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("lacusta.in")));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("lacusta.out")));
    st.nextToken(); m=(int)st.nval;
    st.nextToken(); n=(int)st.nval;
    a=new int[n+1];

    // citesc numai prima linie
    for(j=1;j<=n;j++) { st.nextToken(); a[j]=(int)st.nval; }
    System.out.print(1+" "+1+" --> ");
    a11=a[1];

    // citesc a doua linie
    st.nextToken(); a[1]=(int)st.nval;
    minc=oo;
    jmin=-1;
```

```
for(j=2;j<=n;j++)
{
  aa=a[j];
  st.nextToken(); a[j]=(int)st.nval;
  if(aa+a[j]<minc) {minc=aa+a[j]; jmin=j;}
}
System.out.print(1+" "+jmin+" --> "+2+" "+jmin+" --> ");
minsol=a11+minc;
j0=jmin;

// citesc mai departe cate o linie si ...
for(i=3;i<=m-1;i++)
{
  minc=oo;
  jmin=-1;
  for(j=1;j<=n;j++)
  {
    aa=a[j];
    st.nextToken(); a[j]=(int)st.nval;
    if(j!=j0) if(aa+a[j]<minc) {minc=aa+a[j]; jmin=j;}
  }
  System.out.print((i-1)+" "+jmin+" --> "+i+" "+jmin+" --> ");
  minsol+=minc;
  j0=jmin;
}

//citesc linia m (primele n-1 componente)
minc=oo;
jmin=-1;
for(j=1;j<=n-1;j++)
{
  aa=a[j];
  st.nextToken(); a[j]=(int)st.nval;
  if(aa+a[j]<minc) {minc=aa+a[j]; jmin=j;}
}
System.out.print((m-1)+" "+jmin+" --> "+m+" "+jmin+" --> ");
minsol+=minc;
j0=jmin;

// citesc ultimul element
st.nextToken(); a[n]=(int)st.nval;
minsol+=a[n];
System.out.println(m+" "+n);
```

```
    out.println(minsol);
    out.close();

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1));
  }// main()
}// class
```

16.2.18 Avere ONI2005 cls 10

Italag a fost toată viaţa pasionat de speculaţii bursiere reuşind să adune o avere considerabilă. Fiind un tip original şi pasionat de matematică a scris un testament inedit. Testamentul conţine două numere naturale: S reprezentând averea ce trebuie împărţită moştenitorilor şi N reprezentând alegerea sa pentru împărţirea averii.

Italag decide să-şi împartă toată averea, iar sumele pe care le acordă moştenitorilor să fie în ordine strict descrescătoare.

De exemplu dacă averea ar fi 7 unităţi monetare, ar putea fi împărţită astfel:

• 4 (unităţi primului moştenitor) 3 (unităţi celui de-al doilea), sau

• 6 (unităţi primului moştenitor) 1 (unitate celui de-al doilea), sau

• 7 (unităţi doar primului moştenitor), sau

• 5 (unităţi primului moştenitor) 2 (unităţi celui de-al doilea), sau

• 4 (unităţi primului moştenitor) 2 (unităţi celui de-al doilea) 1 (unitate celui de-al treilea).

Văzând că îi este foarte greu să verifice dacă nu cumva a omis vreo variantă de împărţire, Italag le-a scris în ordine lexicografică. Pentru exemplul de mai sus: 4 2 1; 4 3; 5 2; 6 1; 7.

A hotărât ca banii să fie distribuiţi conform celei de-a N-a posibilităţi din ordinea lexicografică.

Cerinţă

Scrieţi un program care pentru numerele S, N date să calculeze şi să afişeze numărul total de posibilităţi de împărţire a averii, precum şi modul în care se face această împărţire conform cu a N-a posibilitate din ordinea lexicografică.

Datele de intrare

Fişierul de intrare **avere.in** conţine o singură linie pe care se află două numere naturale separate printr-un singur spaţiu:

– primul număr (S) reprezintă suma totală

– cel de-al doilea (N) reprezintă numărul de ordine al poziţiei căutate.

Datele de ieşire

Fişierul de ieşire **avere.out** va conţine două linii:

– pe prima linie va fi afişat numărul total de modalităţi de împărţire a averii;

– pe cea de-a doua linie va fi afişată a N-a posibilitate de împărţire a lui S conform cerinţei în ordine lexicografică. Elementele sale vor fi separate prin câte un spaţiu.

Restricţii şi precizări

• $1 < S < 701$

• $0 < N <$ numărul total de posibilităţi cu suma S

• Se acordă punctaj parţial pentru fiecare test: 5 puncte pentru determinarea corectă a numărului de posibilităţi de împărţire a lui S şi 5 puncte pentru determinarea corectă a posibilităţii N, din ordinea lexicografică.

• Posibilităţile de împărţire a averii sunt numerotate începând cu 1.

• Fie $x = (x_1, x_2..., x_m)$ şi $y = (y_1, y_2, ..., y_p)$ două şiruri. Spunem că x precedă pe y din punct de vedere lexicografic, dacă există $k \geq 1$, astfel încât $x_i = y_i$, pentru orice $i = 1, ..., k-1$ şi $x_k < y_k$.

Exemple: 4 2 1 precedă secvenţa 4 3 deoarece ($4 = 4$, $2 < 3$), iar 6 1 precedă 7 deoarece $6 < 7$.

Exemple:

avere.in	**avere.out**
7 2	5 4 3

avere.in	**avere.out**
12 5	15 6 5 1

avere.in	**avere.out**
700 912345678912345678	962056220379782044 175 68 63 58 54 45 40 36 34 32 20 18 17 14 11 9 3 2 1

Timp maxim de execuţie/test: 1 secundă pentru Windows şi 0.1 secunde pentru Linux.

Limita totală de memorie sub Linux este 3Mb din care 1Mb pentru stivă.

Indicaţii de rezolvare - descriere soluţie *

stud. Emilian Miron, Universitatea Bucureşti

Problema se rezolvă prin *programare dinamică* după valoarea maximă pe care poate să o ia primul număr în cadrul descompunerii şi după suma totală.

Obţinem recurenţa:

$$
\begin{aligned}
c[v][s] &= c[v-1][s] && \text{//punând pe prima poziţie 1, 2, ...,v-1}\\
&\quad +c[v-1][s-v] && \text{//punnd pe prima poziţie v}\\
c[0][0] &= 1, &&\\
c[0][s] &= 0 && \text{pentru } s>0
\end{aligned}
$$

Numărul total de posibilităţi va fi egal cu $c[S][S]$.

Reconstituirea soluţiei se face stabilind primul număr ca fiind cel mai mic i astfel încât $c[i][S] \geq N$ şi $c[i-1][S] < N$. Procesul continuă pentru $S = S - i$ şi $N = N - c[i-1][S]$ până când $N = 0$.

Observăm că recurenţa depinde doar de linia anterioară, aşa că ea se poate calcula folosind un singur vector. Aceasta ne ajută pentru a respecta limita de memorie. Astfel calculăm toate valorile folosind un singur vector şi păstrăm la fiecare pas $c[i][S]$. Reconstrucţia se face recalculând valorile la fiecare pas pentru S-ul curent.

Soluţia descrisă efectuează $O(S^2 * L)$ operaţii, unde L este lungimea soluţiei. Observând $L = maximO(S^{1/2})$ timpul total este $O(S^{3/2})$.

O soluţie backtracking obţine 20 puncte, iar una cu memorie $O(N^2)$ 50 puncte.

Codul sursă

```
import java.io.*;     // c[0][0]=1; c[0][s]=0; pentru s=1,2,...,S
class Avere1          // c[v][s]=c[v-1][s]; pentru v>=1 si s<v
{                     // c[v][s]=c[v-1][s]+c[v-1][s-v]; pentru v>=1 si s>=v
  static int S;       // test 4 ??? test 5 = gresit N (>...) in fisier intrare
  static long n;      // toate celelalte: OK rezultat si timp !!!
  static long c[][];  // dar fara economie de memorie !!!!!!!

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.nanoTime();

    int s,v;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("10-avere.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("avere.out")));
    st.nextToken(); S=(int)st.nval;
    st.nextToken(); n=(int)st.nval;
    c=new long[S+1][S+1];

   for(v=0;v<=S;v++) c[v][0]=(long)1;
```

```
    for(v=1;v<=S;v++)
      for(s=0;s<=S;s++)
        if(s<v) c[v][s]=c[v-1][s]; else c[v][s]=c[v-1][s]+c[v-1][s-v];
    //afism();

    out.println(c[S][S]);
    while((n>0)&&(S>0))
    {
      v=0;
      while(c[v][S]<n) v++;
      out.print(v+" ");
      n=n-c[v-1][S];
      S=S-v;
    }
    out.close();
    t2=System.nanoTime();
    System.out.println("Timp = "+((double)(t2-t1))/1000000000);
  }// main(...)

  static void afism()
  {
    int i,j;
    for(i=0;i<=S;i++)
    {
      for(j=0;j<=S;j++) System.out.print(c[i][j]+" ");
      System.out.println();
    }
  }// afism()
}// class
/*
1 0 0 0 0 0 0 0 0 0  0  0  0
1 1 0 0 0 0 0 0 0 0  0  0  0
1 1 1 1 0 0 0 0 0 0  0  0  0
1 1 1 2 1 1 1 0 0 0  0  0  0
1 1 1 2 2 2 2 2 1 1  1  0  0
1 1 1 2 2 3 3 3 3 3  3  2  2
1 1 1 2 2 3 4 4 4 5  5  5  5
1 1 1 2 2 3 4 5 5 6  7  7  8
1 1 1 2 2 3 4 5 6 7  8  9 10
1 1 1 2 2 3 4 5 6 8  9 10 12
1 1 1 2 2 3 4 5 6 8 10 11 13
1 1 1 2 2 3 4 5 6 8 10 12 14
1 1 1 2 2 3 4 5 6 8 10 12 15
Press any key to continue...
```

```
*/
```

16.2.19 Suma - ONI2005 cls 10

Tradiţia este ca, la ieşirea la pensie, pentru fiecare zi de activitate în slujba sultanului, marele vizir să primească o primă stabilită de marele sfat al ţării. Astfel, vizirul Magir a primit pentru doar 5 zile de activitate prima totală de 411 galbeni, deoarece sfatul ţării a hotărât pentru ziua întâi o sumă de 53 de galbeni, pentru ziua a doua 200 de galbeni, pentru ziua a treia 12 galbeni, pentru ziua a patra 144 de galbeni, iar pentru ziua a cincea doar 2 galbeni.

Vizirul Jibal, celebru pentru contribuţia adusă la rezolvarea conflictului din zonă, primeşte dreptul ca, la ieşirea la pensie, să modifice sumele stabilite de sfatul ţării, dar nu foarte mult. El poate uni cifrele sumelor stabilite şi le poate despărţi apoi, după dorinţă, astfel încât, suma primită pe fiecare zi să nu depăşească 999 de galbeni şi să primească cel puţin un galben pentru fiecare dintre zilele de activitate. Astfel, dacă are doar 5 zile de activitate, plătite cu 23, 417, 205, 5 şi respectiv 40 de galbeni, în total 680 de galbeni, el poate opta pentru o nouă distribuţie a cifrelor numerelor stabilite de marele sfat astfel: pentru prima zi cere 2 galbeni, pentru a doua 3, pentru a treia 417, pentru a patra 205 şi pentru a cincea 540 de galbeni, primind astfel 1167 de galbeni în total.

Cerinţă

Pentru numărul de zile n şi cele n sume stabilite de sfatul ţării pentru Jibal, scrieţi un program care să determine cea mai mare primă totală care se poate obţine prin unirea şi despărţirea cifrelor sumelor date.

Datele de intrare

Fişierul de intrare **suma.in** conţine:

– pe prima linie un număr natural n reprezentând numărul de zile de activitate

– pe linia următoare, n numere naturale separate prin spaţii s_1, s_2, ..., s_n reprezentând sumele atribuite de sfatul ţării.

Datele de ieşire

Fişierul de ieşire **suma.out** va conţine o singură linie pe care va fi afişat un singur număr natural reprezentând prima totală maximă care se poate obţine.

Restricţii şi precizări

- $1 < n < 501$
- $0 < s_i < 1000$, pentru orice $1 \leq i \leq n$
- În orice distribuţie, fiecare sumă trebuie să fie o valoare proprie (să nu înceapă cu 0).
- Orice sumă dintr-o distribuţie trebuie să fie nenulă.
- Pentru 20% din teste, $n \leq 10$, pentru 50% din teste $n \leq 50$.

Exemple

suma.in	**suma.out**	*Explicaţie*
3 58 300 4	362	Prima maximă (362) se obţine chiar pentru distribuţia 58 300 4

suma.in	**suma.out**	*Explicaţie*
5 23 417 205 5 40	1608	Prima maximă (1608) se obţine pentru distribuţia 2 341 720 5 540

Timp maxim de execuţie/test: 1 secundă pentru Windows şi 0.1 secunde pentru Linux.

Indicaţii de rezolvare - descriere soluţie

Soluţia oficială

Soluţia I

Soluţia propusă utilizează *metoda programării dinamice*. Este implementat un *algoritm de expandare tip Lee* realizat cu o coadă alocată dinamic.

Astfel, fiecare cifră contribuie la expandarea soluţiilor precedente care au şansă de dezvoltare ulterioară. Vectorul `best` memorează la fiecare moment suma cea mai mare formată dintr-un număr dat de termeni.

Condiţia `nc-nr<=(n-p^.t-1)*3+2` (unde *nc* este numărul total de cifre care se distribuie, *nr* este numărul de ordine al cifrei curente, *n* este numărul total de termeni şi `p^.t` este numărul de termeni ai soluţiei curente) testează ca, prin crearea unui nou termen cu ajutorul cifrei curente, să mai existe şansa construirii cu cifrele rămase a unei soluţi cu *n* termeni.

Condiţia `nc-nr>=n-p^.t` testează ca, prin lipirea cifrei curente la ultimul termen al soluţiei curente, să mai existe şansa construirii cu cifrele rămase a unei soluţii cu *n* termeni.

```
type pnod=^nod;
     nod=record
       s:longint;   {suma}
       t,last:word; {nr. de termeni si ultimul termen}
       next:pnod
     end;

var n,nc,i,k:longint; f:text;
    best:array[1..1000]of longint;
    p,u:pnod;
    c:char;

procedure citire; {determina numarul total de cifre}
var i,x:longint;
begin
```

```
   assign(f,'suma.in');reset(f);
   readln(f,n);
   for i:=1 to n do begin
     read(f,x);
     repeat inc(nc);x:=x div 10 until x=0
   end;
   close(f)
end;

{expandarea corespunzatoare cifrei curente}
procedure calc(nr:longint;cif:byte);
var c,q:pnod; gata:boolean;
begin
  c:=u;gata:=false;
  repeat
    if (cif>0) and (nc-nr<=(n-p^.t-1)*3+2) and (best[p^.t]=p^.s) then
      begin
        new(u^.next);u:=u^.next;
        u^.s:=p^.s+cif;
        u^.t:=p^.t+1;
        u^.last:=cif
      end;
    if (p^.last<100)and(nc-nr>=n-p^.t) then
      begin
        new(u^.next);u:=u^.next;
        u^.s:=p^.s+p^.last*9+cif;
        u^.t:=p^.t;
        u^.last:=p^.last*10+cif;
      end;
    if p=c then gata:=true;
    q:=p;p:=p^.next;dispose(q)
  until gata;
end;

{recalcularea valorilor maxime memorate in vectorul best}
procedure optim;
var i:longint;
    q:pnod; gata:boolean;
begin
  for i:=1 to n do best[i]:=0;
  q:=p;gata:=false;
  repeat
    if q^.s>best[q^.t] then best[q^.t]:=q^.s;
    if q=u then gata:=true;
```

```
    q:=q^.next
  until gata;
end;

BEGIN
   citire;
   {reluarea citirii cifrelor, ignorand spatiile}
   reset(f); readln(f);
   repeat read(f,c) until c<>' ';
   new(p);
   p^.s:=ord(c)-48;p^.t:=1;
   p^.last:=p^.s;
   best[1]:=p^.s;
   u:=p;
   for i:=2 to nc do begin
     repeat read(f,c) until c<>' ';
       calc(i,ord(c)-48);
       optim
     end;
  close(f);
  assign(f,'suma.out');rewrite(f);writeln(f,best[n]);close(f)
END.
```

Soluţia II

Problema se rezolvă prin metoda *programare dinamică*. Concatenăm numerele şi obţinem un şir (să îl notăm a) de cifre (de lungime L maxim $N * 3$).

Pentru fiecare poziţie p ($p = 1..L$) calculăm prima maximă pe care o putem obţine despărţind subşirul de până la p inclusiv în n sume ($n = 1..N$). Obţinem relaţia:

```
smax[p][n] = min(
  smax[p-1][n-1]+a[p], // punând ultima sumă formată doar din cifra a[i]
  smax[p-2][n-1]+a[p-1]*10+a[p], // punând ultima sumă din 2 cifre
  smax[p-3][n-1]+a[p-2]*100+a[p-1]*10+a[p] // punând ultima sumă din 3 cifre
  )
```

Trebuie avute în vedere cazurile limită când $p = 1$, $p = 2$ sau $p = 3$ şi cazurile în care $a[p]$, $a[p-1]$ sau $a[p-2]$ sunt zero, moment în care nu putem forma o sumă de lungimea respectivă, aşa că excludem termenii din expresia de minim.

Pentru uşurinţa în implementare stocăm $smax[p][n] = -infinit$ pentru cazul în care subşirul de până la p nu poate fi împărţit în mod corect în n sume, iar observând că recurenţa depinde doar de ultimele 3 linii, nu păstrăm decât pe acestea şi linia curentă pentru a nu avea probleme cu memoria.

Obţinem memorie $O(N)$ şi timp de execuţie $O(L * N) = O(N^2)$;

Codul sursă

Varianta 1:

```
import java.io.*; // fara economie de spatiu
class Suma1
{
  static int n,nc; // nc=nr cifre din sir
  static int[] c=new int[1501]; // sirul cifrelor
  static int[][] s;

  public static void main (String[] args) throws IOException
  {
    int i,j,x;
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("suma.in")));
    PrintWriter out=new PrintWriter(new BufferedWriter(
         new FileWriter("suma.out")));
    st.nextToken(); n=(int)st.nval;

    nc=0;
    j=0;
    for(i=1;i<=n;i++)
    {
      st.nextToken(); x=(int)st.nval;
      if(x<10) {c[++j]=x; nc+=1;} else
      if(x<100) {c[++j]=x/10; c[++j]=x%10; nc+=2;} else
      if(x<1000) {c[++j]=x/100; c[++j]=(x/10)%10; c[++j]=x%10; nc+=3;}
      else System.out.println("Eroare date !");
    }

    s=new int[nc+1][n+1];
    calcul();
    afism();
    out.print(s[nc][n]);
    out.close();
  }// main(...)

  static void calcul()
  {
    //  s[i][j]=max(s[i-1][j-1]+c[i],                 // xj are 1: cifra c[i]
    //              s[i-2][j-1]+c[i-1]*10+c[i], // xj are 2 cifre: c[i-1]c[i]
    //              s[i-3][j-1]+c[i-2]*100+c[i-1]*10+c[i]); // xj are 3 cifre
```

```
    int i,j,smax;

    s[1][1]=c[1];
    s[2][1]=c[1]*10+c[2];
    s[3][1]=c[1]*100+c[2]*10+c[3];

    if(c[2]!=0) s[2][2]=c[1]+c[2];

    if((c[2]!=0)&&(c[3]!=0)) s[3][3]=c[1]+c[2]+c[3];

    if(c[3]==0) { if(c[2]!=0) s[3][2]=c[1]+c[2]; }
    else // c[3]!=0
    {
      if(c[2]==0) s[3][2]=c[1]*10+c[3];
      else                                  // c[2]!=0 && c[3]!=0
      s[3][2]=max(c[1]+c[2]*10+c[3],c[1]*10+c[2]+c[3]);
    }

    for(i=4;i<=nc;i++)   // i = pozitie cifra in sirul cifrelor
    for(j=1;j<=n;j++)    // j = pozitie numar in sirul final al numerelor
    {
      smax=0;

      if(j<=i)
      {
        if((c[i]!=0)&&(s[i-1][j-1]!=0))
          smax=max(smax,s[i-1][j-1]+c[i]);

        if((c[i-1]!=0)&&(s[i-2][j-1]!=0))
          smax=max(smax,s[i-2][j-1]+c[i-1]*10+c[i]);

        if((c[i-2]!=0)&&(s[i-3][j-1]!=0))
          smax=max(smax,s[i-3][j-1]+c[i-2]*100+c[i-1]*10+c[i]);
      }

      s[i][j]=smax;
    }// for
  }// calcul()

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }
```

```
  static void afism()
  {
    int i,j;

    System.out.print("     \t");
    for(j=1;j<=n;j++) System.out.print(j+"\t");
    System.out.println();

    for(i=1;i<=nc;i++)
    {
      System.out.print(i+" "+c[i]+" :\t");
      for(j=1;j<=n;j++) System.out.print(s[i][j]+"\t");
      System.out.println();
    }
  }// afism()
}// class
```

Varianta 2:

```
import java.io.*; // cu economie de spatiu !!!
class Suma2
{
  static int n,nc; // nc=nr cifre din sir
  static int[] c=new int[1501]; // sirul cifrelor
  static int[][] s;

  public static void main (String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();

    int i,j,x;
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("suma.in")));
    PrintWriter out=new PrintWriter(new BufferedWriter(
         new FileWriter("suma.out")));
    st.nextToken(); n=(int)st.nval;

    nc=0;
    j=0;
    for(i=1;i<=n;i++)
    {
      st.nextToken(); x=(int)st.nval;
      if(x<10) {c[++j]=x; nc+=1;} else
```

```
    if(x<100) {c[++j]=x/10; c[++j]=x%10; nc+=2;} else
    if(x<1000) {c[++j]=x/100; c[++j]=(x/10)%10; c[++j]=x%10; nc+=3;}
    else System.out.println("Eroare date !");
  }

  s=new int[4][n+1]; // cu economie de spatiu !!!
  calcul();

  out.print(s[nc%4][n]);
  out.close();
  t2=System.currentTimeMillis();
  System.out.println("Timp = "+(t2-t1));
}// main(...)

static void calcul()
{
  //  s[i][j]=max(s[i-1][j-1]+c[i],// xj are 1: cifra c[i]
  //              s[i-2][j-1]+c[i-1]*10+c[i], // xj are 2 cifre: c[i-1]c[i]
  //              s[i-3][j-1]+c[i-2]*100+c[i-1]*10+c[i]); // xj are 3 cifre

  int i,j,smax;

  s[1][1]=c[1];
  s[2][1]=c[1]*10+c[2];
  s[3][1]=c[1]*100+c[2]*10+c[3];

  if(c[2]!=0) s[2][2]=c[1]+c[2];

  if((c[2]!=0)&&(c[3]!=0)) s[3][3]=c[1]+c[2]+c[3];

  if(c[3]==0) { if(c[2]!=0) s[3][2]=c[1]+c[2]; }
  else // c[3]!=0
  {
    if(c[2]==0) s[3][2]=c[1]*10+c[3];
    else // c[2]!=0 && c[3]!=0
    s[3][2]=max(c[1]+c[2]*10+c[3],c[1]*10+c[2]+c[3]);
  }

  for(i=4;i<=nc;i++) // i = pozitie cifra in sirul cifrelor
  for(j=1;j<=n;j++) // j = pozitie numar in sirul final al numerelor
  {
    smax=0;

    if(j<=i)
```

```
      {
        if((c[i]!=0)&&(s[(i-1+4)%4][j-1]!=0))
        smax=max(smax,s[(i-1+4)%4][j-1]+c[i]);

        if((c[i-1]!=0)&&(s[(i-2+4)%4][j-1]!=0))
          smax=max(smax,s[(i-2+4)%4][j-1]+c[i-1]*10+c[i]);

        if((c[i-2]!=0)&&(s[(i-3+4)%4][j-1]!=0))
          smax=max(smax,s[(i-3+4)%4][j-1]+c[i-2]*100+c[i-1]*10+c[i]);
      }

      s[i%4][j]=smax;
    }
  }// calcul()

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }
}// class
```

Varianta 3:

```
import java.io.*; // fara economie de spatiu dar cu afisare o solutie !
class Suma3
{
  static int n,nc; // nc=nr cifre din sir
  static int[] c=new int[1501]; // sirul cifrelor
  static int[][] s;
  static int[][] p; // predecesori
  static int[] sol; // o solutie

  public static void main (String[] args) throws IOException
  {
    int i,j,x;
    StreamTokenizer st=new StreamTokenizer(
          new BufferedReader(new FileReader("suma.in")));
    PrintWriter out=new PrintWriter(new BufferedWriter(
         new FileWriter("suma.out")));
    st.nextToken(); n=(int)st.nval;

    nc=0;
    j=0;
    for(i=1;i<=n;i++)
```

```
    {
      st.nextToken(); x=(int)st.nval;
      if(x<10) {c[++j]=x; nc+=1;} else
      if(x<100) {c[++j]=x/10; c[++j]=x%10; nc+=2;} else
      if(x<1000) {c[++j]=x/100; c[++j]=(x/10)%10; c[++j]=x%10; nc+=3;}
      else System.out.println("Eroare date !");
    }

    s=new int[nc+1][n+1];
    p=new int[nc+1][n+1];
    calcul();
    afism(s); System.out.println();
    afism(p); System.out.println();

    sol=new int[n+1];
    solutia();
    afisv(sol);

    out.print(s[nc][n]);
    out.close();
}// main(...)

  static void solutia()
  {
    int i,i1,i2,k;
    i2=nc;
    for(k=n;k>=1;k--)
    {
      i1=p[i2][k];
      System.out.print(k+" : "+i1+"->"+i2+" ==> ");
      for(i=i1;i<=i2;i++) sol[k]=sol[k]*10+c[i];
      System.out.println(sol[k]);
      i2=i1-1;
    }
  }// solutia()

  static void calcul()
  {
    //  s[i][j]=max(s[i-1][j-1]+c[i],// xj are 1: cifra c[i]
    //              s[i-2][j-1]+c[i-1]*10+c[i], // xj are 2 cifre: c[i-1]c[i]
    //              s[i-3][j-1]+c[i-2]*100+c[i-1]*10+c[i]); // xj are 3 cifre

    int i,j,smax;
```

```
s[1][1]=c[1];
s[2][1]=c[1]*10+c[2];
s[3][1]=c[1]*100+c[2]*10+c[3];
p[1][1]=p[2][1]=p[3][1]=1;

if(c[2]!=0) s[2][2]=c[1]+c[2];

if((c[2]!=0)&&(c[3]!=0)) s[3][3]=c[1]+c[2]+c[3];

if(c[3]==0) { if(c[2]!=0) s[3][2]=c[1]+c[2]; }
else // c[3]!=0
{
  if(c[2]==0) { s[3][2]=c[1]*10+0+c[3]; p[3][2]=3;}
  else // c[2]!=0 && c[3]!=0
  {
    s[3][2]=max(c[1]+c[2]*10+c[3],c[1]*10+c[2]+c[3]);
    if(s[3][2]==c[1]+c[2]*10+c[3]) p[3][2]=2; else p[3][2]=3;
  }
}

for(i=4;i<=nc;i++) // i = pozitie cifra in sirul cifrelor
for(j=1;j<=n;j++) // j = pozitie numar in sirul final al numerelor
{
  smax=0;

  if(j<=i)
  {
    if((c[i]!=0)&&(s[i-1][j-1]!=0))
    {
      smax=max(smax,s[i-1][j-1]+c[i]);
      if(smax==s[i-1][j-1]+c[i]) p[i][j]=i;
    }

    if((c[i-1]!=0)&&(s[i-2][j-1]!=0))
    {
      smax=max(smax,s[i-2][j-1]+c[i-1]*10+c[i]);
      if(smax==s[i-2][j-1]+c[i-1]*10+c[i]) p[i][j]=i-1;
    }

    if((c[i-2]!=0)&&(s[i-3][j-1]!=0))
    {
      smax=max(smax,s[i-3][j-1]+c[i-2]*100+c[i-1]*10+c[i]);
      if(smax==s[i-3][j-1]+c[i-2]*100+c[i-1]*10+c[i]) p[i][j]=i-2;
    }
```

```
      }// if

      s[i][j]=smax;
    }// for
  }// calcul()

  static int max(int a, int b)
  {
    if(a>b) return a; else return b;
  }

  static void afism(int[][] x)
  {
    int i,j;

    System.out.print("     \t");
    for(j=1;j<=n;j++) System.out.print(j+"\t");
    System.out.println();

    for(i=1;i<=nc;i++)
    {
      System.out.print(i+" "+c[i]+" :\t");
      for(j=1;j<=n;j++) System.out.print(x[i][j]+"\t");
      System.out.println();
    }
  }// afism()

  static void afisv(int[] sol)
  {
    int i,sum=0;
    System.out.println();
    for(i=1;i<=n;i++)
    {
      System.out.print(sol[i]+" ");
      sum+=sol[i];
    }
    System.out.println(" ==> "+sum);
  }// afisv()
}// class
```

Capitolul 17

Potrivirea şirurilor

Considerăm un **text** (un şir de caractere) $t = (t_1, t_2, ..., t_n)$ şi un **şablon** (tot un şir de caractere, numit *pattern* în engleză) $p = (p_1, p_2, ..., p_m)$. Considerăm $m \leq n$ şi dorim să determinăm dacă textul t conţine şablonul p, adică, dacă există $0 \leq d \leq n - m$ astfel încât $t_{d+i} = p_i$ pentru orice $1 \leq i \leq m$. Problema *potrivirii şirurilor* constă în determinarea tuturor valorilor d (considerate *deplasamente*) cu proprietatea menţionată.

17.1 Un algoritm ineficient

Pentru fiecare poziţie i cuprinsă între 1 şi $n-m+1$ vom verifica dacă subşirul $(x_i, x_{i+1}, ..., x_{i+m-1})$ coincide cu y.

```
import java.io.*;
class PotrivireSir
{
  static char[] t,p;
  static int n,m;

  public static void main(String[] args) throws IOException
  {
    int i,j;
    String s;

    BufferedReader br=new BufferedReader(
      new FileReader("potriviresir.in"));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("potriviresir.out")));
```

```
    s=br.readLine();
    n=s.length();
    t=new char[n+1];
    for(i=0;i<n;i++) t[i+1]=s.charAt(i);
    System.out.print("t : ");
    afisv(t,1,n);

    s=br.readLine();
    m=s.length();
    p=new char[m+1];
    for(i=0;i<m;i++) p[i+1]=s.charAt(i);
    System.out.print("p : ");
    afisv(p,1,m);
    System.out.println();

    for(i=1;i<=n-m+1;i++)
    {
      for(j=1;j<=m;j++) if(p[j]!=t[i+j-1]) break;
      j--; // ultima pozi\c tie potrivita

      if(j==m) { afisv(t,1,n); afisv(p,i,i+j-1); System.out.println(); }
    }
    out.close();
  }//main()

  static void afisv(char[] x, int i1, int i2)
  {
    int i;
    for(i=1;i<i1;i++) System.out.print(" ");
    for(i=i1;i<=i2;i++) System.out.print(x[i]);
    System.out.println();
  }// afisv(...)
}//class
/*
x : abababaababaababa
y : abaabab

abababaababaababa
    abaabab

abababaababaababa
         abaabab
*/
```

17.2 Un algoritm eficient - KMP

Algoritmul KMP (Knuth-Morris-Pratt) determină potrivirea şirurilor folosind informaţii referitoare la potrivirea subşirului cu diferite deplasamente ale sale.

Pentru

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t:	a	b	a	b	a	b	a	a	b	a	b	a	a	b	a	b	a

şi

	1	2	3	4	5	6	7
p:	a	b	a	a	b	a	b

există două potriviri:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t:	a	b	a	b	a	b	a	a	b	a	b	a	a	b	a	b	a
p:					a	b	a	a	b	a	b						
					1	2	3	4	5	6	7						

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t:	a	b	a	b	a	b	a	a	b	a	b	a	a	b	a	b	a
p:										a	b	a	a	b	a	b	
										1	2	3	4	5	6	7	

Şirul ineficient de încercări este:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t:	a	b	a	b	a	b	a	a	b	a	b	a	a	b	a	b	a
p:	a	b	a	**a**	b	a	b										
p:		**a**	b	a	a	b	a	b									
p:			a	b	a	**a**	b	a	b								
p:				**a**	b	a	a	b	a	b							
p:					a	b	a	a	b	a	b	*					
p:						**a**	b	a	a	b	a	b					
p:							a	**b**	a	a	b	a	b				
p:								a	b	a	**a**	b	a	b			
p:									**a**	b	a	a	b	a	b		
p:										a	b	a	a	b	a	b	*
p:											**a**	b	a	a	b	a	b
											1	2	3	4	5	6	7

Prima nepotrivire din fiecare încercare este evidenţiată prin caracter **boldat** iar soluţiile sunt marcate cu *.

Dorim să avansăm cu mai mult de un pas la o nouă încercare, fără să riscăm să pierdem vreo soluţie!

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t:	a	b	a	b	a	b	a	a	b	a	b	a	a	b	a	b	a
p:	a	b	a	**a**	b	a	b										
p:			a	b	a	**a**	b	a	b								
p:					a	b	a	a	b	a	b	*					
p:										a	b	a	a	b	a	b	*
p:															a	b	a

Notăm prin $t[i..j]$ secvenţa de elemente consecutive $(t_i, ..., t_j)$ (cuprinsă între poziţiile i şi j) din şirul $t = (t_1, t_2, ..., t_n)$.

Să presupunem că suntem la un pas al verificării potrivirii cu un deplasament d şi prima nepotrivire a apărut pe poziţia i din text şi poziţia $j + 1$ din şablon, deci $t[i - j..i - 1] = p[1..j]$ şi $t[i] \neq p[j + 1]$.

Care este cel mai bun deplasament d' pe care trebuie să-l încercam?

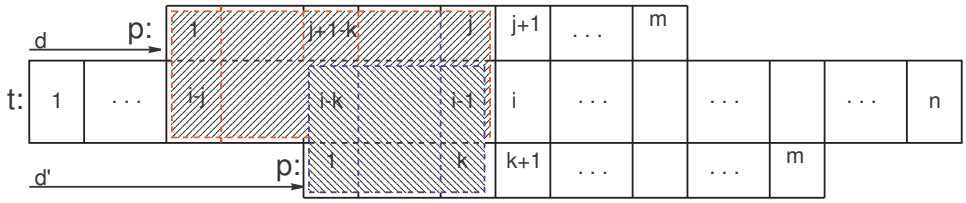


Figura 17.1: Deplasare optimă

Folosind Figura 17.2, dorim să determinăm cel mai mare indice $k < j$ astfel încât $p[1..k] = p[j + 1 - k..j]$. Cu alte cuvinte, dorim să determinăm cel mai lung *sufix* al secvenţei $p[1..j]$ iar noul deplasament d' trebuie ales astfel încât să realizeze acest lucru. Este astfel realizată şi potrivirea textului t cu şablonul p, $t[i - k..i - 1] = p[1..k]$.

Rămâne să verificăm apoi dacă $t[i] = p[k + 1]$.

Observăm că noul deplasament depinde numai de şablonul p şi nu are nici o legătură cu textul t.

Algoritmul KMP utilizează pentru determinarea celor mai lungi sufixe o funcţie numită *next*.

Dat fiind şirul de caractere $p[1..m]$, funcţia

$$next : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m - 1\}$$

este definită astfel:

$$next(j) = \max\{k/k < j \text{ şi } p[1..k] \text{ este sufix pentru } p[1..j]\}.$$

Cum determinăm practic valorile funcţiei *next*?

Initializăm $next[1] = 0$.
Presupunem că au fost determinate valorile $next[1], next[2], ..., next[j]$.
Cum determinăm $next[j+1]$?

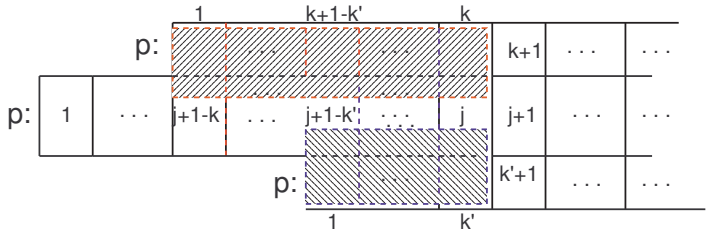


Figura 17.2: Funcţia *next*

Ne ajutăm de Figura 17.2 pentru a urmări mai uşor raţionamentul! În această figură $next[j] = k$ şi $next[k] = k'$.
Dacă $p[j+1] = p[k+1]$ (folosind notaţiile din figură) atunci $next[j+1] = k+1$.
Obţinem:

$$\text{dacă } p[j+1] = p[next(j)+1] \text{ atunci } next[j+1] = next(j)+1.$$

Ce se întâmplă dacă $p[j+1] \neq p[k+1]$?
Căutăm un *sufix* mai mic pentru $p[1..j]$! Fie acesta $p[1..k']$. Dar acest *sufix mai mic* este cel mai lung *sufix* pentru $p[1..k]$, cu alte cuvinte

$$k' = next(k) = next(next(j)).$$

Astfel, dacă $p[j+1] = p[k'+1]$ atunci $next(j+1) = k'+1$.
Obţinem:

$$\text{dacă } p[j+1] = p[next(next(j))+1] \text{ atunci } next[j+1] = next(next(j))+1.$$

Dacă nici acum nu avem egalitate de caractere, vom continua acelaşi raţionament până cand găsim o egalitate de caractere sau lungimea prefixului căutat este 0. Evident, acest algoritm se termină într-un număr finit de paşi pentru că $j > k > k' > ... \geq 0$. Dacă ajungem la 0, atunci vom avea $next(j+1) = 0$.
Ordinul de complexitate al algoritmului KMP este $O(n+m)$.

```
import java.io.*;
class KMP
{
  static int na=0; // nr aparitii
  static char[] t,p; // t[1..n]=text, p[1..m]=pattern
  static int[] next;
```

```
static void readData() throws IOException
{
  String s;
  char[] sc;
  int i,n,m;
  BufferedReader br=new BufferedReader(new FileReader("kmp.in"));
  s=br.readLine();
  sc=s.toCharArray();
  n=sc.length;
  t=new char[n+1];
  for(i=1;i<=n;i++) t[i]=sc[i-1];

  s=br.readLine();
  sc=s.toCharArray();
  m=sc.length;
  p=new char[m+1];
  for(i=1;i<=m;i++) p[i]=sc[i-1];
}//readData()

static int[] calcNext(char[] p)
{
  int m=p.length-1;
  int[] next=new int[m+1]; // next[1..m] pentru p[1..m]
  next[1]=0; // initializare
  int k=0; // nr caractere potrivite
  int j=2;
  while(j<=m)
  {
    while(k>0&&p[k+1]!=p[j]) k=next[k];
    if(p[k+1]==p[j]) k++;
    next[j]=k; j++;
  }
  return next;
}// calcNext()

static void kmp(char[] t, char[] p) // t[1...n], p[1..m]
{
  int n=t.length-1, m=p.length-1;
  next=calcNext(p);
  int j=0; // nr caractere potrivite deja
  int i=1; // ultima pozitie a sufixului
  while(i<=n) // t[1..n]
  {
```

```
      while(j>0&&p[j+1]!=t[i]) j=next[j];
      if(p[j+1]==t[i]) j++;
      if(j==m)
      {
        na++;
        System.out.println("pattern cu deplasarea "+(i-m)+" : ");
        afissol(t,p,i-m);
        j=next[j];
      }
      i++;
    }// while
  }// kmp

  static void afissol(char[] t, char[] p, int d)
  {
    int i, n=t.length-1, m=p.length-1;
    for(i=1;i<=n;i++) System.out.print(t[i]);
    System.out.println();
    for(i=1;i<=d;i++) System.out.print(" ");
    for(i=1;i<=m;i++) System.out.print(p[i]);
    System.out.println();
  }// afissol(...)

  public static void main(String[] args) throws IOException
  {
    readData();
    kmp(t,p);
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("kmp.out")));
    out.println(na);
    out.close();
  }//main()
}//class
/*
pattern apare cu deplasarea 5 :
12312123412123412
     1234
pattern apare cu deplasarea 11 :
12312123412123412
           1234
*/
```

17.3 Probleme rezolvate

17.3.1 Circular - Campion 2003-2004 Runda 6

Autor: prof. Mot Nistor, Colegiul National "N.Balcescu" - Braila

Se spune că şirul $y_1, y_2, ..., y_n$ este o *permutare circulară* cu p poziţii a şirului $x_1, x_2, ..., x_n$ dacă $y_1 = x_p + 1, y_2 = x_p + 2, ..., y_n = x_p + n$, unde indicii mai mari ca n se consideră modulo n, adică indicele k, cu $k > n$ se referă la elementul de indice $k - n$.

Cerinţă

Pentru două şiruri date determinaţi dacă al doilea este o *permutare circulară* a primului şir.

Date de intrare

Pe prima linie a fişierului de intrare **circular.in** este scris numarul natural n. Pe liniile următoare sunt două şiruri de caractere de lungime n, formate numai din litere mari ale alfabetului latin.

Date de ieşire

Pe prima linie a fişierului *circular.out* se va scrie cel mai mic număr natural p pentru care şirul de pe linia a treia este o *permutare circulară* cu p poziţii a şirului de pe linia a doua, sau numărul -1 dacă nu avem o permutare circulară.

Restricţii şi precizări

• $1 \leq n \leq 20000$

Exemple

circular.in	circular.out
10 ABCBAABBAB BABABCBAAB	7

Timp maxim de execuţie/test: 0.1 secunde

Rezolvare (indicaţia autorului): O variantă cu două "for"-uri e foarte uşor de scris, dar nu se încadrează în timp pentru n mare.

Folosim algoritmului KMP de căutare a unui subşir.

Concatenăm primul şir cu el însuşi şi căutăm prima apariţie a celui de-al doilea şir în şirul nou format. În realitate nu e nevoie de *concatenarea* efectivă a şirului, doar ţinem cont că indicii care se referă la şirul "mai lung" trebuie luaţi modulo n.

```
import java.io.*;
class Circular
{
  static int n,d=-1; // pozitia de potrivire
```

```
static char[] x,y; // x[1..n]=text, y[1..m]=pattern
static int[] next;

static void readData() throws IOException
{
  String s;
  char[] sc;
  int i;
  BufferedReader br=new BufferedReader(new FileReader("circular.in"));

  n=Integer.parseInt(br.readLine()); // System.out.println("n="+n);

  x=new char[n+1];
  y=new char[n+1];

  s=br.readLine(); sc=s.toCharArray(); // System.out.println("x="+s);
  for(i=1;i<=n;i++) x[i]=sc[i-1];

  s=br.readLine(); sc=s.toCharArray(); // System.out.println("y="+s);
  for(i=1;i<=n;i++) y[i]=sc[i-1];
}//readData()

static int[] calcNext(char[] p)
{
  int m=n;
  int[] next=new int[m+1];  // next[1..m] pentru p[1..m]
  next[1]=0;                // initializare
  int k=0;                  // nr caractere potrivite
  int j=2;
  while(j<=m)
  {
    while(k>0&&p[k+1]!=p[j]) k=next[k];
    if(p[k+1]==p[j]) k++;
    next[j]=k; j++;
  }
  return next;
}// calcNext()

static void kmp(char[] t, char[] p) // t[1...n], p[1..m]
{
  int m=p.length-1;
  next=calcNext(p);
  int j=0;           // nr caractere potrivite deja
  int i=1;           // ultima pozitie a sufixului
```

```
    while((i<=2*n)&&(d==-1)) // t[1..n]
    {
      while(j>0&&p[j+1]!=t[(i>n)?(i-n):i]) j=next[j];
      if(p[j+1]==t[(i>n)?(i-n):i]) j++;
      if(j==m) { d=i-n; break; }
      i++;
    }// while
  }// kmp

  public static void main(String[] args) throws IOException
  {
    readData();
    kmp(x,y);
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("circular.out")));
    out.println(d);
    out.close();
  }//main()
}//class
/*
circular.in            circular.out
-----------            ------------
20                     5
12312123412123412341
12341212341234112312
*/
```

17.3.2 Cifru - ONI2006 baraj

Copiii solarieni se joacă adesea trimiţându-şi mesaje codificate. Pentru codificare ei folosesc un cifru bazat pe o permutare p a literelor alfabetului solarian şi un număr natural d.

Alfabetul solarian conţine m litere foarte complicate, aşa că noi le vom reprezenta prin numere de la 1 la m.

Dat fiind un mesaj în limbaj solarian, reprezentat de noi ca o succesiune de n numere cuprinse între 1 şi m, $c_1c_2...c_n$, codificarea mesajului se realizează astfel: se înlocuieşte fiecare literă c_i cu $p(c_i)$, apoi şirul obţinut $p(c_1)p(c_2)...p(c_n)$ se roteşte spre dreapta, făcând o permutare circulară cu d poziţii rezultând şirul $p(c_{n-d+1})...p(c_{n-1})p(c_n)p(c_1)p(c_2)...p(c_{n-d})$.

De exemplu, pentru mesajul 213321, permutarea $p = (312)$ (adică $p(1) = 3$, $p(2) = 1$, $p(3) = 2$) şi $d = 2$. Aplicând permutarea p vom obţine şirul 132213, apoi rotind spre dreapta şirul cu două poziţii obţinem codificarea 131322.

Cerinţă:

Date fiind un mesaj necodificat şi codificarea sa, determinaţi cifrul folosit (permutarea p şi numărul d).

Date de intrare:

Fişierul de intrare **cifru.in** conţine pe prima linie numele naturale n şi m, separate prin spaţiu, reprezentând lungimea mesajului şi respectiv numărul de litere din alfabetul solarian. Pe cea de a doua linie este scris mesajul necodificat ca o succesiune de n numere cuprinse între 1 şi m separate prin câte un spaţiu. Pe cea de a treia linie este scris mesajul codificat ca o succesiune de n numere cuprinse între 1 şi m separate prin câte un spaţiu.

Date de ieşire:

Fişierul de ieşire **cifru.out** va conţine pe prima linie numărul natural d, reprezentând numărul de poziţii cu care s-a realizat permutarea circulară spre dreapta. Dacă pentru d există mai multe posibilităţii se va alege valoarea minimă. Pe următoarea linie este descrisă permutarea p. Mai exact se vor scrie valorile $p(1), p(2), ..., p(m)$ separate prin câte un spaţiu.

Restricţii:

$n \leq 100000$

$m \leq 9999$

Mesajul conţine fiecare număr natural din intervalul $[1, m]$ cel puţin o dată.

Pentru teste cu $m \leq 5$ se acordă 40 de puncte din care 20 pentru teste şi cu $n \leq 2000$.

Exemplu:

cifru.in	cifru.out
6 3 2 1 3 3 2 1 1 3 1 3 2 2	2 3 1 2

Timp maxim de execuţie/test: 0.2 secunde

Indicatii de rezolvare:

Soluţia comisiei

Fiecare apariţie a unui simbol din alfabet într-un şir se înlocuieşte cu distanţa faţă de precedenta apariţie a aceluiaşi simbol (considerând şirul circular, deci pentru prima apariţie a simbolului se ia distanţa faţă de ultima apariţie a aceluiaşi simbol).

Făcând această recodificare pentru cele două şiruri reducem problema la determinarea permutării circulare care duce primul şir în al doilea, care poate fi rezolvată cu un algoritm de pattern matching, dacă concatenăm primul şir cu el însuşi rezultând o complexitate $O(n)$.

Pentru m mic se pot genera toate permutările mulţimii $\{1, 2, ..., m\}$ făcând pentru fiecare permutare o căutare (cu KMP de exemplu), iar pentru n mic se poate căuta permutarea pentru fiecare $d = 0, 1, ..., n$.

Codul sursă

```
import java.io.*;
```

```
class kmp
{
  static int[] t0; // text mesaj necodificat --> spatiu ... de eliberat !
  static int[] t1; // text mesaj codificat --> spatiu ... de eliberat !

  static int[] d0; // distante ... mesaj necodificat
  static int[] d1; // distante ... mesaj codificat

  static int[] t;  // text in KMP ... (d0,d0) ... d0 dublat ... spatiu !!!
  static int[] s;  // sablon in KMP ... (d1)
  static int[] p; // prefix in KMP ... 1,2,...n

  static int[] ua; // pozitia ultimei aparitii ... 1,2,...,m ... ==> d[] mai rapid ...
  static int[] perm;// permutarea

  static int n,m; // ... n=100.000, m=9.999 ... maxim !!! ==> 200K

  public static void main(String[] args) throws IOException
  {
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("9-cifru.in")));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("cifru.out")));

    int i,j,j0,j1,k,deplasarea=-1;

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    ua=new int[m+1];

    t0=new int[n+1];
    t1=new int[n+1];
    d0=new int[n+1];
    d1=new int[n+1];
    p=new int[n+1];

    for(i=1;i<=n;i++) { st.nextToken(); t0[i]=(int)st.nval; }
    for(i=1;i<=n;i++) { st.nextToken(); t1[i]=(int)st.nval; }
    distanta(t0,d0);
    distanta(t1,d1);
    //afisv(t0,1,n); afisv(d0,1,n); System.out.println();
    //afisv(t1,1,n); afisv(d1,1,n); System.out.println();
```

```
    s=d0;
    prefix(s,p,n);
    //afisv(s,1,n); afisv(p,1,n); System.out.println();

    t=new int[2*n+1]; // ocupa spatiu prea mult; aici standard dar ...
    for(i=1;i<=n;i++) t[i]=t[n+i]=d1[i];
    //afisv(t,1,2*n);

    deplasarea=kmp(t,2*n,s,n)-1; // d1 dublat si caut d0 ...
    out.println(deplasarea);
    System.out.println(deplasarea);

    // permutarea ...
    perm=ua;            // economie de spatiu ...
    for(i=1;i<=m;i++) perm[i]=0;
    k=0;     // nr elemente plasate deja in permutare ...
    j1=0;
    for(i=1;i<=n;i++)
    {
      j1++;
      j0=n-deplasarea+i;
      if(j0>n) j0=j0-n;
      //System.out.println(i+" : "+j0+" "+j1);
      if(perm[t0[j0]]==0)
      {
        perm[t0[j0]]=t1[j1];
        k++;
      }
      if(k==m) break;
    }
    //afisv(perm,1,m);

    for(i=1;i<=m;i++) out.print(perm[i]+" ");
    out.close();
  }// main

  static int kmp(int[] t, int n, int[] s, int m)// t1,...,tn si s1,...,sm
  {
    int k,i,pozi=-1;
    k=0;
    for (i=1;i<=n;i++)
    {
      while(k>0&&s[k+1]!=t[i]) k=p[k];
      if (s[k+1]==t[i]) k++;
```

```
      if(k==m)
      {
        pozi=i-m+1;
        //System.out.println("incepe pe pozitia "+pozi);
        break; // numai prima aparitie ... !!!
      }
    }// for
    return pozi;
  }// kmp(...)

  static void distanta(int[] t,int[] d) // t=text, d=distante ...
  {
    int i,j,k;
    for(i=1;i<=m;i++) ua[i]=0;

    for(i=1;i<=n;i++)
    {
      if(ua[t[i]]!=0) // stiu pozitia spre stanga a lui t[i] ...
      {
        if(ua[t[i]]<i)
          d[i]=i-ua[t[i]];  // e mai la stanga ...
        else
          d[i]=i-ua[t[i]]+n; // e mai la dreapta ...

        ua[t[i]]=i; // noua pozitie a lui t[i] ...
        continue;
      }

      // nu a aparut inca in 1..i-1 ==> de la n spre stanga
      k=i; // distanta spre stanga ... pana la n inclusiv ...
      j=n; // caut in zona n,n-1,n-2,...
      while(t[i]!=t[j])
      {
        k++;
        j--;
      }
      d[i]=k;
      ua[t[i]]=i;
    }// for i
  }// distanta(...)

  static void prefix(int[] s,int[] p,int m) // s=sablon, p=prefix, m=dimensiune
  {
    int i,k;
```

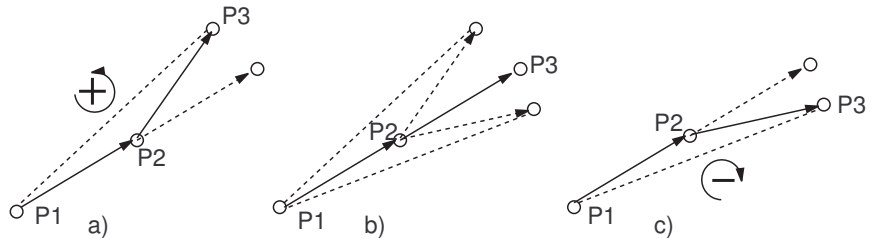
```
    p[1]=0;
    for(i=2;i<=m;i++)
    {
      k=p[i-1];
      while(k>0&&s[k+1]!=s[i]) k=p[k];
      if(s[k+1]==s[i]) p[i]=k+1; else p[i]=0;
    }
  }// prefix()

  static void afisv(int[] x, int i1, int i2)
  {
    int i;
    for(i=i1;i<=i2;i++) System.out.print(x[i]+" ");
    System.out.println();
  }// afisv(...)
}// class
```

Capitolul 18

Geometrie computaţională

18.1 Determinarea orientării

Considerăm trei puncte în plan $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ şi $P_3(x_3, y_3)$.
Panta segmentului P_1P_2: $m_{12} = (y_2 - y_1)/(x_2 - x_1)$
Panta segmentului P_2P_3: $m_{23} = (y_3 - y_2)/(x_3 - x_2)$



Orientarea parcurgerii laturilor P_1P_2 şi P_2P_3 (în această ordine):

- în sens trigonometric (spre stânga): $m_{12} < m_{23}$, cazul a) în figură
- în sensul acelor de ceas (spre dreapta): $m_{12} > m_{23}$, cazul c) în figură
- vârfuri coliniare: $m_{12} = m_{23}$, cazul b) în figură

Orientarea depinde de valoarea expresiei

$$o(P_1(x_1, y_1), P_2(x_2, y_2), P_3(x_3, y_3)) = (y_2 - y_1) \cdot (x_3 - x_2) - (y_3 - y_2) \cdot (x_2 - x_1)$$

astfel

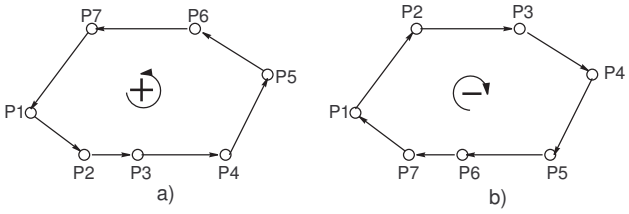
$$o(P_1(x_1,y_1),P_2(x_2,y_2),P_3(x_3,y_3))\begin{cases}<0 & \Rightarrow \text{ sens trigonometric}\\ =0 & \Rightarrow \text{ coliniare}\\ >0 & \Rightarrow \text{ sensul acelor de ceas}\end{cases}$$

18.2 Testarea convexităţii poligoanelor

Considerăm un poligon cu n vârfuri $P_1(x_1,y_1)P_2(x_2,y_2)...P_n(x_ny_n)$, $n \geq 3$. Poligonul este convex dacă şi numai dacă perechile de segmente

$$(P_1P_2,P_2P_3),(P_2P_3,P_3P_4),...,(P_{n-2}P_{n-1},P_{n-1}P_n) \text{ şi } (P_{n-1}P_n,P_nP_1)$$

au aceeaşi orientare sau sunt colineare.



18.3 Aria poligoanelor convexe

Aria poligonului convex cu n vârfuri $P_1(x_1,y_1)P_2(x_2,y_2)...P_n(x_ny_n)$, $n \geq 3$ se poate determina cu ajutorul următoarei formule:

$$\frac{1}{2}\left|x_1y_2+x_2y_3+...+x_{n-1}y_n+x_ny_1-y_1x_2+y_2x_3+...+y_{n-1}x_n+y_nx_1\right|$$

Expresia de sub modul este pozitivă dacă orientarea $P_1 \to P_2 \to ...P_n \to P_1$ este în sens trigonometric, este negativă dacă orientarea $P_1 \to P_2 \to ...P_n \to P_1$ este în sensul acelor de ceasornic şi este nulă dacă punctele $P_1(x_1,y_1)$, $P_2(x_2,y_2)$, ..., $P_n(x_ny_n)$ sunt colineare. Reciproca acestei afirmaţii este deasemenea adevărată în cazul poligoanelor convexe.

18.4 Poziţia unui punct faţă de un poligon convex

Considerăm un poligon convex cu n vârfuri $P_1(x_1, y_1)P_2(x_2, y_2)...P_n(x_n y_n)$, $n \geq 3$ şi un punct $P_0(x_0, y_0)$. Dorim să determinăm dacă punctul $P_0(x_0, y_0)$ este în interiorul poligonului.

Pentru comoditatea prezentării considerăm şi punctul $P_{n+1}(x_{n+1}, y_{n+1})$ unde $x_1 = x_{n+1}$ şi $y_1 = y_{n+1}$, adică punctul P_{n+1} este de fapt tot punctul P_1.

Considerăm o latură oarecare $[P_i P_{i+1}]$ $(1 \leq i \leq n)$ a poligonului.

Ecuaţia dreptei $(P_i P_{i+1})$ este

$$(P_i P_{i+1}) : y - y_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} (x - x_i)$$

Aducem la acelaşi numitor şi considerăm funcţia

$$f_i(x, y) = (y - y_i) \cdot (x_{i+1} - x_i) - (x - x_i) \cdot (y_{i+1} - y_i)$$

Dreapta $(P_i P_{i+1})$ împarte planul în două semiplane. Funcţia $f_i(x, y)$ are valori de acelaşi semn pentru toate punctele din acelaşi semiplan, valori cu semn contrar pentru toate punctele din celălalt semiplan şi valoarea 0 pentru doate punctele situate pe dreaptă.

Pentru a fi siguri că punctul $P_0(x_0, y_0)$ se află în interiorul poligonului (acesta fiind convex) trebuie să verificăm dacă toate vârfurile poligonului împreună cu punctul $P_0(x_0, y_0)$ sunt de aceeaşi parte a dreptei $(P_i P_{i+1})$, adică toate valorile $f_i(x_j, y_j)$ $(1 \leq j \leq n,\ j \neq i$ şi $j \neq i + 1)$ au acelaşi semn cu $f_i(x_0, y_0)$ (sau sunt nule dacă acceptăm prezenţa punctului $P_0(x_0, y_0)$ pe frontiera poligonului). Aceasta este o condiţie necesară dar nu şi suficientă. Vom verifica dacă pentru orice latură $[P_i P_{i+1}]$ $(1 \leq i \leq n)$ a poligonului toate celelalte vârfuri sunt în acelaşi semiplan cu $P_0(x_0, y_0)$ (din cele două determinate de dreapta suport a laturii respective) iar dacă se întâmplă acest lucru atunci putem trage concluzia că punctul $P_0(x_0, y_0)$ se află în interiorul poligonului convex.

O altă modalitate de verificare dacă punctul $P_0(x_0, y_0)$ este în interiorul sau pe frontiera poligonului convex $P_1(x_1, y_1)P_2(x_2, y_2)...P_n(x_n y_n)$ este verificarea următoarei relaţii:

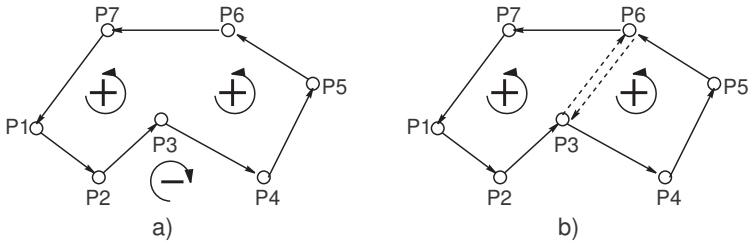
$$arie_{poligon}(P_1 P_2 ... P_n) = \sum_{k=1}^{n} arie_{triunghi}(P_0 P_k P_{k+1})$$

unde punctul $P(x_{n+1}, y_{n+1})$ este de fapt tot punctul $P_1(x_1, y_1)$.

18.5 Poziţia unui punct faţă de un poligon concav

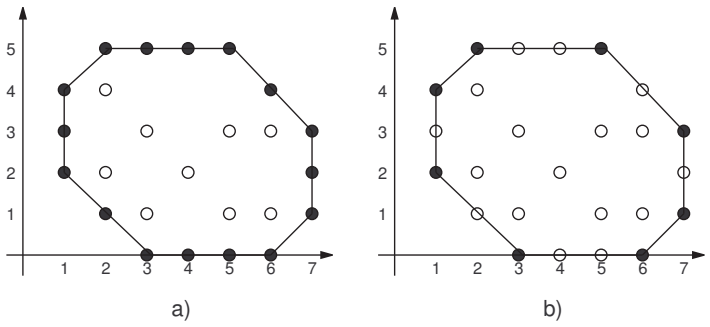
Considerăm un poligon concav cu n vârfuri $P_1(x_1, y_1)P_2(x_2, y_2)...P_n(x_n y_n)$, $n \geq 3$ şi un punct $P_0(x_0, y_0)$. Dorim să determinăm dacă punctul $P_0(x_0, y_0)$ este în interiorul poligonului.

Poligonul concav se descompune în poligoane convexe cu ajutorul diagonalelor *interne* şi se foloseşte un algoritm pentru poligoane convexe pentru fiecare poligon convex astfel obţinut. Dacă punctul este în interiorul unui poligon convex obţinut prin partiţionarea poligonului concav atunci el se află în interiorul acestuia. Dacă nu se află în nici un poligon convex obţinut prin partiţionarea poligonului concav atunci el nu se află în interiorul acestuia.



18.6 Înfăşurătoarea convexă

18.6.1 Împachetarea Jarvis



Toate punctele de pe înfăşurătoarea convexă (cazul a) în figură):

```
import java.io.*; // infasuratoare convexa
class Jarvis1     // pe frontiera coliniare ==> le iau pe toate ... !!!
{
```

```
static int n,npic=0;  // npic=nr puncte pe infasuratoarea convexa
static int [] x;
static int [] y;
static int [] p;      // precedent
static int [] u;      // urmator

static void afisv(int[] a, int k1, int k2)
{
  int k;
  for(k=k1;k<=k2;k++) System.out.print(a[k]+" ");
  System.out.println();
}

static int orient(int i1, int i2, int i3)
{
  long s=(y[i1]-y[i2])*(x[i3]-x[i2])-(y[i3]-y[i2])*(x[i1]-x[i2]);
  if(s<0) return -1; else
  if(s>0) return 1; else return 0;
}

static void infasurareJarvis() throws IOException
{
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

  int i0,i,i1,i2;
  i0=1;
  for(i=2;i<=n;i++) if((x[i]<x[i0])||((x[i]==x[i0])&&(y[i]<y[i0]))) i0=i;
  System.out.println("Stanga_Jos ==> P"+i0+" : "+x[i0]+" "+y[i0]);

  i1=i0;
  npic++;
  System.out.println(npic+" --> "+i1); //br.readLine();
  do
  {
    i2=i1+1; if(i2>n) i2-=n;
    for(i=1;i<=n;i++)
    {
      //System.out.println("orient("+i1+","+i+","+i2+")="+orient(i1,i,i2));
      //br.readLine();
      if(orient(i1,i,i2)>0) i2=i; else
        if(orient(i1,i,i2)==0)  // coliniare
          if(                   // i intre i1 i2 ==> cel mai apropiat
          ((x[i]-x[i1])*(x[i]-x[i2])<0)||
          ((y[i]-y[i1])*(y[i]-y[i2])<0)
```

```
            )
            i2=i;
      }
      u[i1]=i2;
      p[i2]=i1;
      i1=i2;
      npic++;
      System.out.println(npic+" --> "+i1); //br.readLine();
    } while(i2!=i0);
    npic--; // apare de doua ori primul punct !
    System.out.print("u : "); afisv(u,1,n);
    System.out.print("p : "); afisv(p,1,n);
  }// infasurareJarvis()

  public static void main(String[] args) throws IOException
  {
    int k;
    StreamTokenizer st= new StreamTokenizer(
      new BufferedReader(new FileReader("jarvis.in")));
    st.nextToken(); n=(int)st.nval;
    x=new int[n+1];
    y=new int[n+1];
    p=new int[n+1];
    u=new int[n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
    }
    infasurareJarvis();
  }//main
}//class
```

Fără punctele coliniare de pe înfăşurătoarea convexă (cazul *b*) în figură):

```
import java.io.*; // infasuratoare convexa
class Jarvis2     // pe frontiera coliniare ==> iau numai capetele ... !!!
{
  static int n,npic=0;  // npic=nr puncte pe infasuratoarea convexa
  static int [] x;
  static int [] y;
  static int [] p;     // precedent
  static int [] u;     // urmator
```

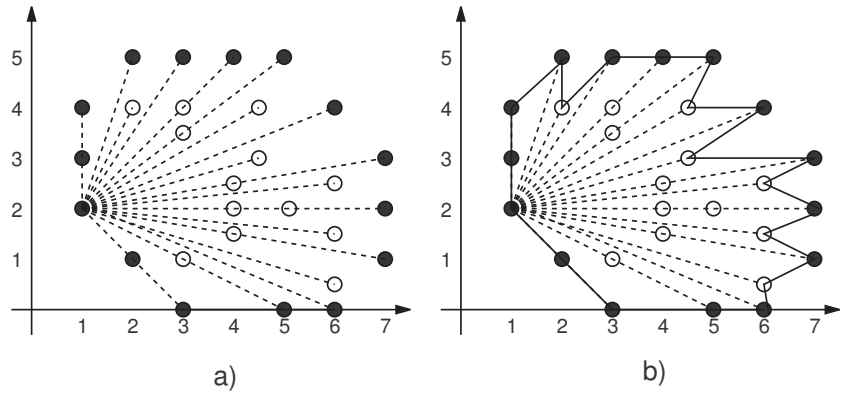
```
static void afisv(int[] a, int k1, int k2)
{
  int k;
  for(k=k1;k<=k2;k++) System.out.print(a[k]+" ");
  System.out.println();
}

static int orient(int i1, int i2, int i3)
{
  long s=(y[i1]-y[i2])*(x[i3]-x[i2])-(y[i3]-y[i2])*(x[i1]-x[i2]);
  if(s<0) return -1; else
  if(s>0) return 1; else return 0;
}

static void infasurareJarvis() throws IOException
{
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

  int i0,i,i1,i2;
  i0=1;
  for(i=2;i<=n;i++) if((x[i]<x[i0])||((x[i]==x[i0])&&(y[i]<y[i0]))) i0=i;
  System.out.println("Stanga_Jos ==> P"+i0+" : "+x[i0]+" "+y[i0]);

  i1=i0;
  npic++;
  System.out.println(npic+" --> "+i1); //br.readLine();
  do
  {
    i2=i1+1; if(i2>n) i2-=n;
    for(i=1;i<=n;i++)
    {
      //System.out.println("orient("+i1+","+i+","+i2+")="+orient(i1,i,i2));
      //br.readLine();
      if(orient(i1,i,i2)>0) i2=i; else
      if(orient(i1,i,i2)==0)  // coliniare
      if(                     // i2 intre i1 i ==> cel mai departat
         ((x[i2]-x[i1])*(x[i2]-x[i])<0)||
         ((y[i2]-y[i1])*(y[i2]-y[i])<0)
         )
        i2=i;
    }
    u[i1]=i2;
    p[i2]=i1;
    i1=i2;
```

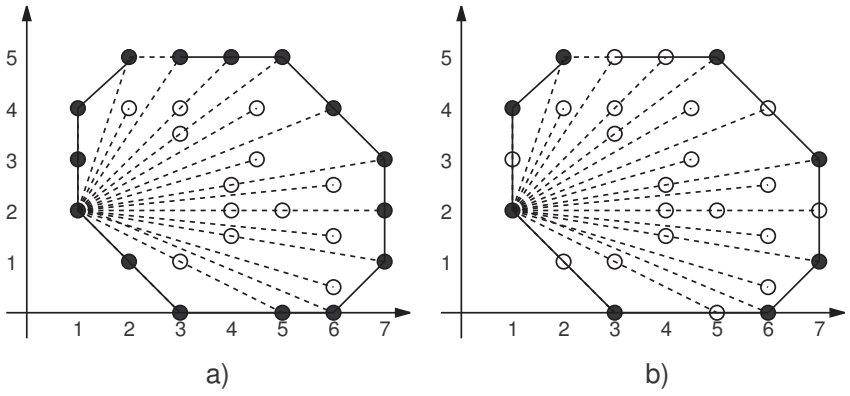
```
      npic++;
      System.out.println(npic+" --> "+i1); //br.readLine();
    } while(i2!=i0);
    npic--; // apare de doua ori primul punct !
    System.out.print("u : "); afisv(u,1,n);
    System.out.print("p : "); afisv(p,1,n);
  }// infasurareJarvis()

  public static void main(String[] args) throws IOException
  {
    int k;
    StreamTokenizer st= new StreamTokenizer(
      new BufferedReader(new FileReader("jarvis.in")));
    st.nextToken(); n=(int)st.nval;
    x=new int[n+1];
    y=new int[n+1];
    p=new int[n+1];
    u=new int[n+1];
    for(k=1;k<=n;k++)
    {
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
    }
    infasurareJarvis();
  }//main
}//class
```

18.6.2 Scanarea Craham

Versiune cu mesaje pentru sortarea punctelor:

```
import java.io.*; // numai pentru sortare si mesaje ...
class Graham0
{
  static int n,npic=0; // npic=nr puncte pe infasuratoarea convexa
  static int[] x;
  static int[] y;
  static int[] o; // o[k]  = pozitia lui k inainte de sortare
  static int[] of; // of[k] = pozitia lui k dupa sortare

  // pentru depanare ... stop ! ...
  static BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```






```
static void afisv(int[] a, int k1, int k2)
{
  int k;
  for(k=k1;k<=k2;k++) System.out.print(a[k]+" ");
  System.out.print("  ");
}

static int orient(int i0,int i1, int i2)
{
  long s=(y[i1]-y[i0])*(x[i2]-x[i0])-(y[i2]-y[i0])*(x[i1]-x[i0]);
  if(s<0) return -1; else
  if(s>0) return 1; else return 0;
}
```

```
static void qsort(int p, int u) throws IOException
{
  // aleg un punct fix k
  int k=(p+u)/2;

  System.out.println("qsort: p="+p+" u="+u+" k="+k+" xk="+x[k]+" yk="+y[k]);
  System.out.print("x : "); afisv(x,p,u); System.out.println();
  System.out.print("y : "); afisv(y,p,u); System.out.println();

  int i,j,aux;
  i=p;j=u;
  while(i<j)
  {
    while( (i<j)&&
           (  (orient(1,i,k)<0)||
              ((orient(1,i,k)==0)&&
              (((x[i]-x[1])*(x[i]-x[k])<0)||((y[i]-y[1])*(y[i]-y[k])<0)))
           )
         )
    {
      i++;
    }
    while(  (i<j)&&
            (  (orient(1,j,k)>0)||
               ((orient(1,j,k)==0)&&
               (((x[j]-x[1])*(x[j]-x[k])>0)||((y[j]-y[1])*(y[j]-y[k])>0)))
            )
         )
    {
      j--;
    }

    if(i<j)
    {
      if(k==i) k=j; else if(k==j) k=i;// k=fix dar se poate schimba pozitia
      aux=x[i]; x[i]=x[j]; x[j]=aux;
      aux=y[i]; y[i]=y[j]; y[j]=aux;
      aux=o[i]; o[i]=o[j]; o[j]=aux;
    }
  }// while

  System.out.println("Final while ... i="+i+"  j="+j);
  // i=j si P[i] este pe locul lui !!!
```

```
    System.out.print("x : "); afisv(x,p,i-1); afisv(x,i,i); afisv(x,i+1,u);
    System.out.println();
    System.out.print("y : "); afisv(y,p,i-1); afisv(y,i,i); afisv(y,i+1,u);
    System.out.println();
    br.readLine();

    if(p<i-1) qsort(p,i-1);
    if(j+1<u) qsort(j+1,u);
  }// qSort(...)

  static void scanareGraham() throws IOException
  {
    int i0,i,i1,i2,aux;

    System.out.print("x : "); afisv(x,1,n); System.out.println();
    System.out.print("y : "); afisv(y,1,n); System.out.println();
    System.out.println();

    i0=1;
    for(i=2;i<=n;i++) if((x[i]<x[i0])||((x[i]==x[i0])&&(y[i]<y[i0]))) i0=i;
    System.out.println("Stanga_Jos ==> P"+i0+" : "+x[i0]+" "+y[i0]+"\n");

    aux=x[1]; x[1]=x[i0]; x[i0]=aux;
    aux=y[1]; y[1]=y[i0]; y[i0]=aux;
    aux=o[1]; o[1]=o[i0]; o[i0]=aux;

    System.out.print("x : "); afisv(x,1,n); System.out.println();
    System.out.print("y : "); afisv(y,1,n); System.out.println();
    System.out.print("o : "); afisv(o,1,n); System.out.println();
    System.out.println();

    qsort(2,n);

    System.out.println();
    System.out.print("x : "); afisv(x,1,n); System.out.println();
    System.out.print("y : "); afisv(y,1,n); System.out.println();
    System.out.print("o : "); afisv(o,1,n); System.out.println();
    System.out.println();
  }// scanareGraham()

  public static void main(String[] args) throws IOException
  {
    int k;
```

```
    StreamTokenizer st= new StreamTokenizer(
      new BufferedReader(new FileReader("graham.in")));
    st.nextToken(); n=(int)st.nval;
    x=new int[n+1];
    y=new int[n+1];
    o=new int[n+1];
    of=new int[n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
      o[k]=k;
    }

    scanareGraham();

    // ordinea finala (dupa sortare) a punctelor
    for(k=1;k<=n;k++) of[o[k]]=k;
    System.out.println();
    System.out.print("of : "); afisv(of,1,n); System.out.println();
    System.out.println();
  }//main
}//class
```

Versiune cu toate punctele de pe înfăşurătoare:

```
import java.io.*; // NU prinde punctele coliniarele pe ultima latura !
class Graham1     // daca schimb ordinea pe "razele" din sortare, atunci ...
{                 // NU prinde punctele coliniarele pe prima latura, asa ca ...
  static int n;
  static int np; // np=nr puncte pe infasuratoarea convexa
  static int[] x;
  static int[] y;
  static int[] o; // pozitia inainte de sortare
  static int[] p; // poligonul infasuratoare convexa

  static int orient(int i0,int i1, int i2)
  {
    long s=(y[i1]-y[i0])*(x[i2]-x[i0])-(y[i2]-y[i0])*(x[i1]-x[i0]);
    if(s<0) return -1; else
    if(s>0) return 1; else return 0;
  }

  static void qsort(int p, int u)
```

```
{
  int i,j,k,aux;
  // aleg un punct fix k
  k=(p+u)/2;
  i=p;
  j=u;
  while(i<j)
  {
    while(  (i<j)&&
            (  (orient(1,i,k)<0)||
               ((orient(1,i,k)==0)&&
               (((x[i]-x[1])*(x[i]-x[k])<0)||((y[i]-y[1])*(y[i]-y[k])<0)))
            )
         ) i++;
    while(  (i<j)&&
            (  (orient(1,j,k)>0)||
               ((orient(1,j,k)==0)&&
               (((x[j]-x[1])*(x[j]-x[k])>0)||((y[j]-y[1])*(y[j]-y[k])>0)))
            )
         ) j--;
    if(i<j)
    {
      if(k==i) k=j; else if(k==j) k=i;// k=fix dar se poate schimba pozitia
      aux=x[i]; x[i]=x[j]; x[j]=aux;
      aux=y[i]; y[i]=y[j]; y[j]=aux;
      aux=o[i]; o[i]=o[j]; o[j]=aux;
    }
  }
  if(p<i-1) qsort(p,i-1);
  if(j+1<u) qsort(j+1,u);
}// qSort(...)

static void scanareGraham() throws IOException
{
  int i0,i,i1,i2,i3,aux;

  i0=1;
  for(i=2;i<=n;i++) if((x[i]<x[i0])||((x[i]==x[i0])&&(y[i]<y[i0]))) i0=i;

  aux=x[1]; x[1]=x[i0]; x[i0]=aux;
  aux=y[1]; y[1]=y[i0]; y[i0]=aux;
  aux=o[1]; o[1]=o[i0]; o[i0]=aux;

  qsort(2,n);
```

```
    i1=1; p[1]=i1;
    i2=2; p[2]=i2;
    np=2;

    i3=3;
    while(i3<=n)
    {
      while(orient(i1,i2,i3)>0)
      {
        i2=p[np-1];
        i1=p[np-2];
        np--;
      }
      np++;
      p[np]=i3;
      i2=p[np];
      i1=p[np-1];
      i3++;
    }// while

    // plasez si punctele coliniare de pe ultima latura a infasuratorii
    i=n-1;
    while(orient(1,p[np],i)==0) p[++np]=i--;

    // afisez rezultatele
    System.out.print("punctele initiale: ");
    for(i=1;i<=np;i++) System.out.print(o[p[i]]+" ");
    System.out.println();
    System.out.print("infasuratoare x: ");
    for(i=1;i<=np;i++) System.out.print(x[p[i]]+" ");
    System.out.println();
    System.out.print("infasuratoare y: ");
    for(i=1;i<=np;i++) System.out.print(y[p[i]]+" ");
    System.out.println();
  }// scanareGraham()

  public static void main(String[] args) throws IOException
  {
    int k;

    StreamTokenizer st= new StreamTokenizer(
      new BufferedReader(new FileReader("graham1.in")));
    st.nextToken(); n=(int)st.nval;
```

```
    x=new int[n+1];
    y=new int[n+1];
    o=new int[n+1];
    p=new int[n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
      o[k]=k;
    }

    scanareGraham();

  }//main
}//class
```

Versiune fără puncte coliniare pe înfăşurătoare:

```
import java.io.*; // aici ... infasuratoarea nu contine puncte coliniare ...
class Graham2     // este o eliminare din rezultatul final dar ...
{                 // se pot elimina puncte la sortare si/sau scanare ...
  static int n;
  static int np; // np=nr puncte pe infasuratoarea convexa
  static int[] x;
  static int[] y;
  static int[] o; // pozitia inainte de sortare
  static int[] p; // poligonul infasuratoare convexa

  static int orient(int i0,int i1, int i2)
  {
    long s=(y[i1]-y[i0])*(x[i2]-x[i0])-(y[i2]-y[i0])*(x[i1]-x[i0]);
    if(s<0) return -1; else
    if(s>0) return 1; else return 0;
  }

  static void qsort(int p, int u)// elimin si punctele coliniare (din interior)
  {
    int i,j,k,aux;
    // aleg un punct fix k
    k=(p+u)/2;
    i=p;
    j=u;
    while(i<j)
    {
```

```
      while(  (i<j)&&
              (  (orient(1,i,k)<0)||
                 ((orient(1,i,k)==0)&&
                 (((x[i]-x[1])*(x[i]-x[k])<0)||((y[i]-y[1])*(y[i]-y[k])<0)))
              )
           ) i++;
      while(  (i<j)&&
              (  (orient(1,j,k)>0)||
                 ((orient(1,j,k)==0)&&
                 (((x[j]-x[1])*(x[j]-x[k])>0)||((y[j]-y[1])*(y[j]-y[k])>0)))
              )
           ) j--;
      if(i<j)
      {
        if(k==i) k=j; else if(k==j) k=i;// k=fix dar se poate schimba pozitia
        aux=x[i]; x[i]=x[j]; x[j]=aux;
        aux=y[i]; y[i]=y[j]; y[j]=aux;
        aux=o[i]; o[i]=o[j]; o[j]=aux;
      }
    }
    if(p<i-1) qsort(p,i-1);
    if(j+1<u) qsort(j+1,u);
  }// qSort(...)

  static void scanareGraham() throws IOException
  {
    int i0,i,i1,i2,i3,aux;

    i0=1;
    for(i=2;i<=n;i++) if((x[i]<x[i0])||((x[i]==x[i0])&&(y[i]<y[i0]))) i0=i;

    aux=x[1]; x[1]=x[i0]; x[i0]=aux;
    aux=y[1]; y[1]=y[i0]; y[i0]=aux;
    aux=o[1]; o[1]=o[i0]; o[i0]=aux;

    qsort(2,n);

    i1=1; p[1]=i1;
    i2=2; p[2]=i2;
    np=2;

    i3=3;
    while(i3<=n)
    {
```

```
      while(orient(i1,i2,i3)>0) // elimin i2
      {
        i2=p[np-1];
        i1=p[np-2];
        np--;
      }
      np++;
      p[np]=i3;
      i2=p[np];
      i1=p[np-1];
      i3++;
    }// while

    // eliminarea punctelor coliniare de pe infasuratoare
    p[np+1]=p[1];
    for(i=1;i<=np-1;i++)
      if(orient(p[i],p[i+1],p[i+2])==0) o[p[i+1]]=0;

    // afisez rezultatele
    System.out.print("punctele initiale: ");
    for(i=1;i<=np;i++) if(o[p[i]]!=0) System.out.print(o[p[i]]+" ");
    System.out.println();
    System.out.print("infasuratoare x: ");
    for(i=1;i<=np;i++) if(o[p[i]]!=0) System.out.print(x[p[i]]+" ");
    System.out.println();
    System.out.print("infasuratoare y: ");
    for(i=1;i<=np;i++) if(o[p[i]]!=0) System.out.print(y[p[i]]+" ");
    System.out.println();
  }// scanareGraham()

  public static void main(String[] args) throws IOException
  {
    int k;

    StreamTokenizer st= new StreamTokenizer(
      new BufferedReader(new FileReader("graham2.in")));
    st.nextToken(); n=(int)st.nval;
    x=new int[n+2];
    y=new int[n+2];
    o=new int[n+2];
    p=new int[n+2];

    for(k=1;k<=n;k++)
    {
```

```
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
      o[k]=k;
    }

    scanareGraham();

  }//main
}//class
```

18.7 Dreptunghi minim de acoperire a punctelor

Se poate determina dreptunghiul minim de acoperire pentru înfăşurătoarea convexă (figura 18.1) pentru a prelucra mai puţine puncte dar nu este obligatorie această strategie.

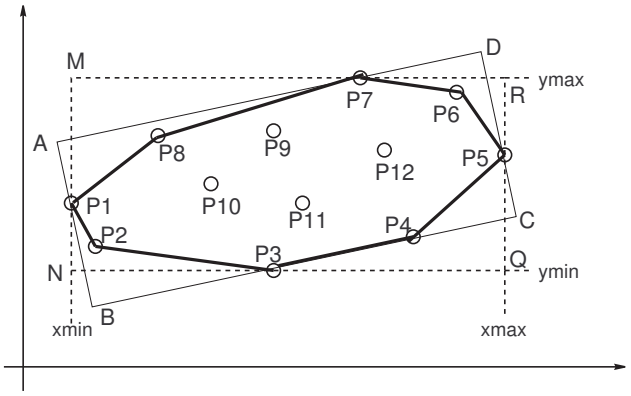
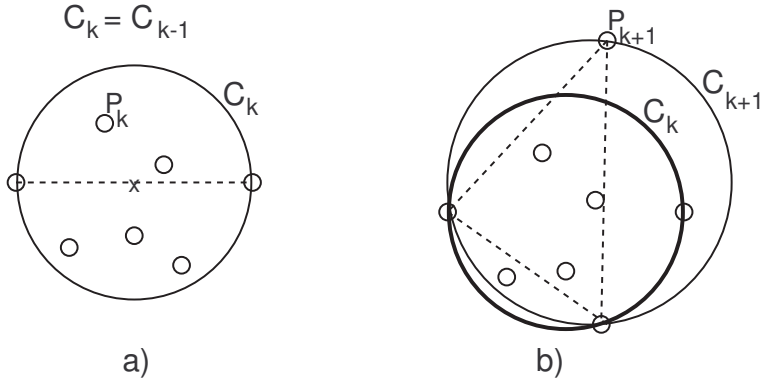


Figura 18.1: Dreptunghi minim de acoperire

Putem să presupunem că punctele formează un poligon convex. Determinarea dreptunghiului de arie minimă care conţine în interiorul său (inclusiv frontiera) toate punctele date se poate face observând că o latură a sa conţine o latură a poligonului convex. Pentru fiecare latură a poligonului convex se determină dreptunghiul minim de acoperire care conţine acea latură. Dintre aceste dreptunghiuri se alege cel cu aria minimă.

18.8 Cerc minim de acoperire a punctelor

Se poate determina cercul minim de acoperire pentru înfăşurătoarea convexă pentru a prelucra mai puţine puncte dar nu este obligatorie această strategie.



Ordonăm punctele astfel încât pe primele poziţii să fie plasate punctele de extrem (cel mai din stânga, urmat de cel mai din dreapta, urmat de cel mai de jos, urmat de cel mai de sus; după acestea urmează celelalte puncte într-o ordine oarecare). Presupunem că punctele, după ordonare, sunt: $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, $P_3(x_3, y_3)$, ..., $P_n(x_n, y_n)$.

Notăm cu $C_i(a_i, b_i; r_i)$ cercul de centru (a_i, b_i) şi rază minimă r_i care acoperă punctele $P_1, P_2, ..., P_n$.

Considerăm cercul $C_2(a_2, b_2; r_2)$ unde $a_2 = (x_1 + x_2)/2$, $b_2 = (y_1 + y_2)/2$ şi $r_2 = \frac{1}{2}\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, adică cercul de diametru $[P_1P_2]$.

Să presupunem că am determinat, pas cu pas, cercurile C_2, C_3, ..., C_i şi trebuie să determinăm cercul C_{i+1}.

Dacă punctul P_{i+1} se află în interiorul cercului C_i atunci cercul C_{i+1} este identic cu C_i.

Dacă punctul P_{i+1} nu se află în interiorul cercului C_i atunci cercul C_{i+1} se determină reluînd algoritmul pentru şirul de puncte P_1, P_2, ...,P_i, P_{i+1} dar impunând condiţia ca acest cerc să treacă în mod obligatoriu prin punctul $P_{i+1}(x_{i+1}, y_{i+1})$. Putem plasa acest punct pe prima poziţie în şirul punctelor şi astfel vom impune la fiecare pas ca punctul P_1 să fie pe cercul care trebuie determinat!

18.9 Probleme rezolvate

18.9.1 Seceta - ONI2005 clasa a IX-a

lect. Ovidiu Domşa

Grădinile roditoare ale Bărăganului suferă anual pierderi imense din cauza secetei. Căutătorii de apă au găsit n fântâni din care doresc să alimenteze n grădini. Fie G_i, F_i, $i = 1, ..., n$ puncte în plan reprezentând puncte de alimentare ale grădinilor şi respectiv punctele în care se află fântânile. Pentru fiecare punct se dau coordonatele întregi (x, y) în plan.

Pentru a economisi materiale, legătura dintre o grădină şi o fântână se realizează printr-o conductă în linie dreaptă. Fiecare fântână alimentează o singură grădină. Consiliul Judeţean Galaţi plăteşte investiţia cu condiţia ca lungimea totală a conductelor să fie minimă.

Fiecare unitate de conductă costă 100 lei noi (RON).

Cerinţă

Să se determine m, costul minim total al conductelor ce leagă fiecare grădină cu exact o fântână.

Date de intrare

Fişierul de intrare **seceta.in** va conţine:

• Pe prima linie se află numărul natural n, reprezentând numărul grădinilor şi al fântânilor.

• Pe următoarele n linii se află perechi de numere întregi G_x G_y, separate printr-un spaţiu, reprezentând coordonatele punctelor de alimentare ale grădinilor.

• Pe următoarele n linii se află perechi de numere întregi F_x F_y, separate printr-un spaţiu, reprezentând coordonatele punctelor fântânilor.

Date de ieşire

Fişierul de ieşire **seceta.out** va conţine:

m − un număr natural reprezentând partea întreagă a costului minim total al conductelor.

Restricţii şi precizări

• $1 < n < 13$

• $0 \leq Gx, Gy, Fx, Fy \leq 200$

• Nu există trei puncte coliniare, indiferent dacă sunt grădini sau fântâni

• Orice linie din fişierele de intrare şi ieşire se termină prin marcajul de sfârşit de linie.

Exemplu

seceta.in	**seceta.out**	Explicaţie
3	624	Costul minim este [6.24264 * 100]=624
1 4		prin legarea perechilor:
3 3		Gradini Fantani
4 7		1 4 2 3
2 3		3 3 3 1
2 5		4 7 2 5
3 1		

Timp maxim de execuţie/test: 1 sec sub Windows şi 0.5 sec sub Linux.

Indicaţii de rezolvare *

Soluţia oficială, lect. Ovidiu Domşa

Numărul mic al punctelor permite generarea tuturor posibilităţilor de a conecta o grădină cu o fântână neconectată la un moment dat.

Pentru fiecare astfel de combinaţie găsită se calculează suma distanţelor (Gi, Fj), în linie dreapta, folosind formula distanţei dintre două puncte în plan, studiată la geometrie. $(d(A(x,y), B(z,t) = \sqrt{(x-z)^2 + (y-t)^2})$.

Acestă soluţie implementată corect asigură $60 - 70$ de puncte.

Pentru a obţine punctajul maxim se tine cont de următoarele aspecte:

1. Se construieşte în prealabil matricea distanţelor $d(i,j)$ cu semnificaţia distanţei dintre grădina i şi fântâna j. Aceasta va reduce timpul de calcul la variantele cu peste 9 perechi.

2. Pentru a elimina cazuri care nu pot constitui soluţii optime se foloseşte proprietatea patrulaterului că suma a doua laturi opuse (condiţie care asigură unicitatea conectării unei singure fântâni la o singură grădină) este mai mică decât suma diagonalelor. De aceea nu se vor lua în considerare acele segmente care se intersectează. Condiţia de intersecţie a două segmente care au capetele în punctele de coordonate $A(a1, a2)$, $B(b1, b2)$, $C(c1, c2)$, $D(d1, d2)$ este ca luând segmentul AB, punctele C şi D să se afle de aceeaşi parte a segmentului AB şi respectiv pentru segmentul CD, punctele A şi B să se afle de aceeaşi parte (se înlocuieşte în ecuaţia dreptei ce trece prin două puncte, studiată în clasa a 9-a).

Observaţie: Pentru cei interesaţi, problema are soluţie şi la un nivel superior, folosind algoritmul de determinare a unui flux maxim de cost minim.

Variantă cu determinarea intesecţiei segmentelor.

```
import java.io.*; // cu determinarea intesectiei segmentelor
class Seceta1     // Java este "mai incet" decat Pascal si C/C++
{                 // test 9 ==> 2.23 sec
  static int nv=0;
  static int n;
  static int[] xg, yg, xf, yf, t, c;
  static int[] a; // permutare: a[i]=fantana asociata gradinii i
  static double costMin=200*1.42*12*100;
  static double[][] d;
  static PrintWriter out;
  static StreamTokenizer st;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    citire();
```

```
    rezolvare();
    afisare();
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }

  static void citire() throws IOException
  {
    int k;
    st=new StreamTokenizer(new BufferedReader(new FileReader("seceta.in")));
    st.nextToken(); n=(int)st.nval;

    xg=new int[n+1];
    yg=new int[n+1];
    xf=new int[n+1];
    yf=new int[n+1];
    a=new int[n+1];
    d=new double[n+1][n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); xg[k]=(int)st.nval;
      st.nextToken(); yg[k]=(int)st.nval;
    }
    for(k=1;k<=n;k++)
    {
      st.nextToken(); xf[k]=(int)st.nval;
      st.nextToken(); yf[k]=(int)st.nval;
    }
  }

  static void rezolvare()  throws IOException
  {
    int i,j;
    int s;
    for(i=1;i<=n;i++) // gradina i
      for(j=1;j<=n;j++) // fantana j
      {
        s=(xg[i]-xf[j])*(xg[i]-xf[j])+(yg[i]-yf[j])*(yg[i]-yf[j]);
        d[i][j]=Math.sqrt(s);
      }
    f(1); // generez permutari
  }
```

```
static void f(int k)
{
  boolean ok;
  int i,j;
  for(i=1;i<=n;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;
    for(j=1;j<k;j++)
      if(seIntersecteaza(xg[k],yg[k],xf[i],   yf[i],
                         xg[j],yg[j],xf[a[j]],yf[a[j]]))
      {
        ok=false;
        break;
      }
    if(!ok) continue;
    a[k]=i;
    if(k<n) f(k+1); else verificCostul();
  }
}

static void verificCostul()
{
  int i;
  double s=0;
  for(i=1;i<=n;i++) s=s+d[i][a[i]];
  if(s<costMin) costMin=s;
}

// de ce parte a dreptei [(xa,ya);(xb,yb)] se afla (xp,yp)
static int s(int xp,int yp,int xa,int ya,int xb,int yb)
{
  double s=(double)yp*(xb-xa)-xp*(yb-ya)+xa*yb-xb*ya;
  if(s<-0.001) return -1;      // in zona "negativa"
    else if(s>0.001) return 1; // in zona "pozitiva"
         else return 0;        // pe dreapta suport
}

// testeaza daca segmentul[P1,P1] se intersecteaza cu [P3,P4]
static boolean seIntersecteaza(int x1, int y1, int x2, int y2,
                               int x3, int y3, int x4, int y4)
{
  double x,y;
```

```
    if((x1==x2)&&(x3==x4))     // ambele segmente verticale
      if(x1!=x3) return false;
      else if(intre(y1,y3,y4)||intre(y2,y3,y4)) return true;
           else return false;

    if((y1==y2)&&(y3==y4))     // ambele segmente orizontale
      if(y1!=y3) return false;
      else if(intre(x1,x3,x4)||intre(x2,x3,x4)) return true;
           else return false;

    if((y2-y1)*(x4-x3)==(y4-y3)*(x2-x1))    // au aceeasi panta (oblica)
      if((x2-x1)*(y3-y1)==(y2-y1)*(x3-x1))  // au aceeasi dreapta suport
        if(intre(x1,x3,x4)||intre(x2,x3,x4)) return true;
        else return false;
      else return false;// nu au aceeasi dreapta suport
    else // nu au aceeasi panta (macar unul este oblic)
    {
      x=(double)((x4-x3)*(x2-x1)*(y3-y1)-
                 x3*(y4-y3)*(x2-x1)+
                 x1*(y2-y1)*(x4-x3))/
                ((y2-y1)*(x4-x3)-(y4-y3)*(x2-x1));
      if(x2!=x1) y=y1+(y2-y1)*(x-x1)/(x2-x1); else y=y3+(y4-y3)*(x-x3)/(x4-x3);
      if(intre(x,x1,x2)&&intre(y,y1,y2)&&intre(x,x3,x4)&&intre(y,y3,y4))
        return true; else return false;
    }
  }

  static boolean intre(int c, int a, int b) // c este in [a,b] ?
  {
    int aux;
    if(a>b) {aux=a; a=b; b=aux;}
    if((a<=c)&&(c<=b)) return true; else return false;
  }

  static boolean intre(double c, int a, int b) // c este in [a,b] ?
  {
    int aux;
    if(a>b) {aux=a; a=b; b=aux;}
    if((a<=c)&&(c<=b)) return true; else return false;
  }

  static void afisare() throws IOException
  {
    int k;
```

```
    out=new PrintWriter(new BufferedWriter(new FileWriter("seceta.out")));
    out.println((int)(costMin*100));
    out.close();
  }
}
```

Variantă cu cu determinarea pozitiei punctelor in semiplane şi mesaje pentru depanare.

```
import java.io.*;  // cu determinarea pozitiei punctelor in semiplane
class Seceta2      // cu mesaje pentru depanare !
{
  static int nv=0;
  static int n;
  static int[] xg, yg, xf, yf, t, c;
  static int[] a; // permutare: a[i]=fantana asociata gradinii i
  static double costMin=200*1.42*12*100;
  static double[][] d;
  static PrintWriter out;
  static StreamTokenizer st;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    citire();
    rezolvare();
    afisare();
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }

  static void citire() throws IOException
  {
    int k;
    st=new StreamTokenizer(new BufferedReader(new FileReader("seceta.in")));
    st.nextToken(); n=(int)st.nval;
    xg=new int[n+1];
    yg=new int[n+1];
    xf=new int[n+1];
    yf=new int[n+1];
    a=new int[n+1];
    d=new double[n+1][n+1];

    for(k=1;k<=n;k++)
```

```
    {
      st.nextToken(); xg[k]=(int)st.nval;
      st.nextToken(); yg[k]=(int)st.nval;
    }
    for(k=1;k<=n;k++)
    {
      st.nextToken(); xf[k]=(int)st.nval;
      st.nextToken(); yf[k]=(int)st.nval;
    }
  }

  static void rezolvare()  throws IOException
  {
    int i,j;
    int s;
    for(i=1;i<=n;i++) // gradina i
    for(j=1;j<=n;j++) // fantana j
    {
      s=(xg[i]-xf[j])*(xg[i]-xf[j])+(yg[i]-yf[j])*(yg[i]-yf[j]);
      d[i][j]=Math.sqrt(s);
    }
    f(1); // generez permutari
  }

  static void f(int k)
  {
    boolean ok;
    int i,j;
    for(i=1;i<=n;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;

      for(j=1;j<k;j++)
        if((s(xg[k],yg[k],xg[j],yg[j],xf[a[j]],yf[a[j]])*
            s(xf[i],yf[i],xg[j],yg[j],xf[a[j]],yf[a[j]])<0)&&
           (s(xg[j],   yg[j],   xg[k],yg[k],xf[i],yf[i])*
            s(xf[a[j]],yf[a[j]],xg[k],yg[k],xf[i],yf[i])<0))
        {
          afisv(k-1);// pe pozitia k(gradina) vreau sa pun i(fantana)
          System.out.print(i+"  ");// pe pozitia j(gradina) e pus a[j](fantana)
          System.out.print(k+""+i+" "+j+""+a[j]);
          System.out.print("  ("+xg[k]+","+yg[k]+") "+"  ("+xf[i]+","+yf[i]+") ");
```

```
          System.out.println("  ("+xg[j]+","+yg[j]+") "+"  ("+xf[a[j]]+","+yf[a[j]]+") ");

          ok=false;
          break;
        }
      if(!ok) continue;

      a[k]=i;
      if(k<n) f(k+1); else verificCostul();
    }
  }

  static void verificCostul()
  {
    int i;
    double s=0;
    for(i=1;i<=n;i++) s=s+d[i][a[i]];
    if(s<costMin) costMin=s;
    afisv(n); System.out.println(" "+s+" "+costMin+" "+(++nv));
  }

  static void afisv(int nn)
  {
    int i;
    for(i=1;i<=nn;i++) System.out.print(a[i]);
  }

  // de ce parte a dreptei [(xa,ya);(xb,yb)] se afla (xp,yp)
  static int s(int xp,int yp,int xa,int ya,int xb,int yb)
  {
    double s=(double)yp*(xb-xa)-xp*(yb-ya)+xa*yb-xb*ya;
    if(s<-0.001) return -1;     // in zona "negativa"
    else if(s>0.001) return 1;  // in zona "pozitiva"
         else return 0;         // pe dreapta suport
  }

  static void afisare() throws IOException
  {
    int k;
    out=new PrintWriter(new BufferedWriter(new FileWriter("seceta.out")));
    out.println((int)(costMin*100));
    out.close();
  }
}
```

Variantă cu cu determinarea pozitiei punctelor in semiplane, fără mesaje pentru depanare.

```
import java.io.*;   // cu determinarea pozitiei punctelor in semiplane
class Seceta3       // Java este "mai incet" decat Pascal si C/C++
{                   // test 9 ==> 2.18 sec
  static int n;
  static int[] xg, yg, xf, yf, t, c;
  static int[] a; // permutare: a[i]=fantana asociata gradinii i
  static double costMin=200*1.42*12*100;
  static double[][] d;
  static PrintWriter out;
  static StreamTokenizer st;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
    citire();
    rezolvare();
    afisare();
    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }

  static void citire() throws IOException
  {
    int k;
    st=new StreamTokenizer(new BufferedReader(
                           new FileReader("seceta.in")));
    st.nextToken(); n=(int)st.nval;
    xg=new int[n+1];
    yg=new int[n+1];
    xf=new int[n+1];
    yf=new int[n+1];
    a=new int[n+1];
    d=new double[n+1][n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); xg[k]=(int)st.nval;
      st.nextToken(); yg[k]=(int)st.nval;
    }
    for(k=1;k<=n;k++)
```

```
    {
      st.nextToken(); xf[k]=(int)st.nval;
      st.nextToken(); yf[k]=(int)st.nval;
    }
  }

  static void rezolvare()  throws IOException
  {
    int i,j;
    int s;
    for(i=1;i<=n;i++) // gradina i
    for(j=1;j<=n;j++) // fantana j
    {
      s=(xg[i]-xf[j])*(xg[i]-xf[j])+(yg[i]-yf[j])*(yg[i]-yf[j]);
      d[i][j]=Math.sqrt(s);
    }
    f(1); // generez permutari
  }

  static void f(int k)
  {
    boolean ok;
    int i,j;
    for(i=1;i<=n;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;
      for(j=1;j<k;j++)
      if((s(xg[k],   yg[k],   xg[j],yg[j],xf[a[j]],yf[a[j]])*
          s(xf[i],   yf[i],   xg[j],yg[j],xf[a[j]],yf[a[j]])<0)&&
         (s(xg[j],   yg[j],   xg[k],yg[k],xf[i],   yf[i])*
          s(xf[a[j]],yf[a[j]],xg[k],yg[k],xf[i],   yf[i])<0))
      {
        ok=false;
        break;
      }
      if(!ok) continue;
      a[k]=i;
      if(k<n) f(k+1); else verificCostul();
    }
  }

  static void verificCostul()
```

```
  {
    int i;
    double s=0;
    for(i=1;i<=n;i++) s=s+d[i][a[i]];
    if(s<costMin) costMin=s;
  }

  //de ce parte a dreptei [(xa,ya);(xb,yb)] se afla (xp,yp)
  static int s(int xp,int yp,int xa,int ya,int xb,int yb)
  {
    double s=(double)yp*(xb-xa)-xp*(yb-ya)+xa*yb-xb*ya;
    if(s<-0.001) return -1;       // in zona "negativa"
    else  if(s>0.001) return 1;   // in zona "pozitiva"
          else return 0;          // pe dreapta suport
  }

  static void afisare() throws IOException
  {
    int k;
    out=new PrintWriter(new BufferedWriter(new FileWriter("seceta.out")));
    out.println((int)(costMin*100));
    out.close();
  }
}
```

Varianta 4:

```
import java.io.*;   // gresit (!) dar ... obtine 100p ... !!!
class Seceta4       // test 9 : 2.18 sec --> 0.04 sec
{
  static int n;
  static int[] xg, yg, xf, yf, t, c;
  static int[] a;      // permutare: a[i]=fantana asociata gradinii i
  static double costMin=200*1.42*12*100;
  static double[][] d;
  static boolean[] epus=new boolean[13];

  static PrintWriter out;
  static StreamTokenizer st;

  public static void main(String[] args) throws IOException
  {
    long t1,t2;
    t1=System.currentTimeMillis();
```

```
    citire();
    rezolvare();
    afisare();

    t2=System.currentTimeMillis();
    System.out.println("Timp = "+(t2-t1)+" ms");
  }// main(...)

  static void citire() throws IOException
  {
    int k;
    st=new StreamTokenizer(new BufferedReader(new FileReader("seceta.in")));
    st.nextToken(); n=(int)st.nval;
    xg=new int[n+1];
    yg=new int[n+1];
    xf=new int[n+1];
    yf=new int[n+1];
    a=new int[n+1];

    d=new double[n+1][n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); xg[k]=(int)st.nval;
      st.nextToken(); yg[k]=(int)st.nval;
    }
    for(k=1;k<=n;k++)
    {
      st.nextToken(); xf[k]=(int)st.nval;
      st.nextToken(); yf[k]=(int)st.nval;
    }
  }// citire(...)

  static void rezolvare()  throws IOException
  {
    int i,j;
    int s;
    for(i=1;i<=n;i++)      // gradina i
      for(j=1;j<=n;j++)    // fantana j
      {
        s=(xg[i]-xf[j])*(xg[i]-xf[j])+(yg[i]-yf[j])*(yg[i]-yf[j]);
        d[i][j]=Math.sqrt(s);
      }
    f(1);                 // generez permutari
```

```
  }// rezolvare(...)

  static void f(int k)
  {
    int i,j;
    boolean seIntersecteaza;
    for(i=1;i<=n;i++)
    {
      if(epus[i]) continue;
      seIntersecteaza=false;
      for(j=1;j<=k-1;j++)
        if(d[k][i]+d[j][a[j]]>d[j][i]+d[k][a[j]])
        {
          seIntersecteaza=true;
          break;
        }

      if(seIntersecteaza) continue;

      a[k]=i;

      epus[i]=true;
      if(k<n) f(k+1); else verificCostul();
      epus[i]=false;
    }// for i
  }// f(...)

  static void verificCostul()
  {
    int i;
    double s=0;
    for(i=1;i<=n;i++) s=s+d[i][a[i]];
    if(s<costMin) costMin=s;
  }// verificCostul(...)

  static void afisare() throws IOException
  {
    int k;
    out=new PrintWriter(new BufferedWriter(new FileWriter("seceta.out")));
    out.println((int)(costMin*100));
    out.close();
  }// afisare(...)
}// class
```

18.9.2 Antena - ONI2005 clasa a X-a

prof. Osman Ay, Liceul International de Informatică Bucureşti

În Delta Dunării există o zonă sălbatică, ruptă de bucuriile şi necazurile civilizaţiei moderne.

În această zonă există doar n case, poziţiile acestora fiind specificate prin coordonatele carteziene de pe hartă.

Postul de radio al ONI 2005 doreşte să emită pentru toţi locuitorii din zonă şi, prin urmare, va trebui să instaleze o antenă de emisie specială pentru aceasta.

O antenă emite unde radio într-o zonă circulară. Centrul zonei coincide cu punctul în care este poziţionată antena. Raza zonei este denumită puterea antenei. Cu cât puterea antenei este mai mare, cu atât antena este mai scumpă.

Prin urmare trebuie selectată o poziţie optimă de amplasare a antenei, astfel încât fiecare casă să se afle în interiorul sau pe frontiera zonei circulare în care emite antena, iar puterea antenei să fie minimă.

Cerinţă

Scrieţi un program care să determine o poziţie optimă de amplasare a antenei, precum şi puterea minimă a acesteia.

Datele de intrare

Fişierul de intrare **antena.in** conţine pe prima linie un număr natural n, reprezentând numărul de case din zonă. Pe următoarele n linii se află poziţiile caselor. Mai exact, pe linia $i + 1$ se află două numere întregi separate printr-un spaţiu x y, ce reprezintă abscisa şi respectiv ordonata casei i. Nu există două case în aceeaşi locaţie.

Datele de ieşire

Fişierul de ieşire **antena.out** conţine pe prima linie două numere reale separate printr-un spaţiu x y reprezentnd abscisa şi ordonata poziţiei optime de amplasare a antenei.

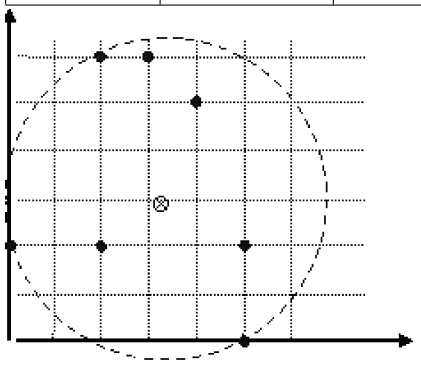
Pe cea de a doua linie se va scrie un număr real reprezentând puterea antenei.

Restricţii şi precizări

- $2 < N < 15001$
- $-15000 < x, y < 15001$
- Numerele reale din fişierul de ieşire trebuie scrise cu trei zecimale cu rotunjire.
- La evaluare, se verifică dacă diferenţa dintre soluţia afişată şi cea corectă (în valoare absolută) este < 0.01.

Exemplu

antena.in	**antena.out**	*Explicaţie*
7 5 0 2 6 4 5 2 2 0 2 3 6 5 2	3.250 2.875 3.366	Antena va fi plasată în punctul de coordonate $(3.250, 2.825)$ iar puterea antenei este 3.366



Casele, antena şi zona acoperită de aceasta

Timp maxim de execuţie/test: 0.3 secunde pentru Windows şi 0.1 secunde pentru Linux.

```
import java.io.*;   // practic, trebuie sa determinam cele trei puncte
class Antena        // prin care trece cercul care le acopera pe toate!!!
{
  static int n;
  static int[] x,y;
  static double x0, y0, r0;

  public static void main(String[] args) throws IOException
  {
    int k;
    long t1,t2;
    t1=System.nanoTime();
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("antena.in")));
    PrintWriter out=new PrintWriter(
                    new BufferedWriter(new FileWriter("antena.out")));
```

```
    st.nextToken(); n=(int)st.nval;
    x=new int[n+1];
    y=new int[n+1];

    for(k=1;k<=n;k++)
    {
      st.nextToken(); x[k]=(int)st.nval;
      st.nextToken(); y[k]=(int)st.nval;
    }

    if(n>3)
    {
      puncteExtreme();
      cercDeDiametru(x[1],y[1],x[2],y[2]);
      for(k=3;k<=n;k++)
      if(!esteInCerc(k))
        cercPrin(x[k],y[k],k-1);   // trece prin Pk si acopera 1,2,...,k-1
    }
    else cercCircumscris(x[1],y[1],x[2],y[2],x[3],y[3]);

    // scriere cu 3 zecimale rotunjite
    out.print( (double)((int)((x0+0.0005)*1000))/1000+" ");
    out.println((double)((int)((y0+0.0005)*1000))/1000);
    out.println((double)((int)((r0+0.0005)*1000))/1000);
    out.close();
    t2=System.nanoTime();
    System.out.println("Timp = "+((double)(t2-t1))/1000000000);
  }// main(...)

  // trece prin (xx,yy) si acopera punctele 1,2,...,k
  static void cercPrin(int xx, int yy, int k)
  {
    int j;
    cercDeDiametru(x[1],y[1],xx,yy);   // trece prin P1 si (xx,yy)

    for(j=2;j<=k;j++)
      if(!esteInCerc(j))
        cercPrin(xx,yy,x[j],y[j],j-1); // ... acopera 1,2,...,j-1
  }// cercPrin(...)

  // trece prin (xx,yy) si (xxx,yyy) si acopera 1,2,3,...,j
  static void cercPrin(int xx,int yy,int xxx,int yyy,int j)
  {
    int i;
```

```
    cercDeDiametru(xx,yy,xxx,yyy);
    for(i=1;i<=j;i++)      // acopera 1,2,...,j
      if(!esteInCerc(i))
        cercCircumscris(xx,yy,xxx,yyy,x[i],y[i]);
  }// cercPrin(...)

  static boolean esteInCerc(int k)
  {
    if(d(x[k],y[k],x0,y0)<r0+0.0001) return true; else return false;
  }

  static void puncteExtreme()
  {
    int k,aux,min,max,kmin,kmax;

    // caut cel mai din stanga punct (si mai jos) si-l pun pe pozitia 1
    // (caut incepand cu pozitia 1)
    kmin=1; min=x[1];
    for(k=2;k<=n;k++)
      if((x[k]<min)||(x[k]==min)&&(y[k]<y[kmin])) {min=x[k]; kmin=k;}
    if(kmin!=1) swap(1,kmin);

    // caut cel mai din dreapta (si mai sus) punct si-l pun pe pozitia 2
    // (caut incepand cu pozitia 2)
    kmax=2; max=x[2];
    for(k=3;k<=n;k++)
      if((x[k]>max)||(x[k]==max)&&(y[k]>y[kmax])) {max=x[k]; kmax=k;}
    if(kmax!=2) swap(2,kmax);

    // caut cel mai de jos (si mai la dreapta) punct si-l pun pe pozitia 3
    // (caut incepand cu pozitia 3)
    kmin=3; min=y[3];
    for(k=4;k<=n;k++)
      if((y[k]<min)||(y[k]==min)&&(x[k]>x[kmin])) {min=y[k]; kmin=k;}
    if(kmin!=3) swap(3,kmin);

    // caut cel mai de sus (si mai la stanga) punct si-l pun pe pozitia 4
    // (caut incepand cu pozitia 4)
    kmax=4; max=y[4];
    for(k=5;k<=n;k++)
      if((y[k]>max)||(y[k]==max)&&(x[k]<x[kmax])) {max=y[k]; kmax=k;}
    if(kmax!=4) swap(4,kmax);

    if(d(x[1],y[1],x[2],y[2])<d(x[3],y[3],x[4],y[4])) // puncte mai departate
```

```
    {
      swap(1,3);
      swap(2,4);
    }
}// puncteExtreme()

  static void cercCircumscris(int x1,int y1,int x2,int y2,int x3,int y3)
  { // consider ca punctele nu sunt coliniare !
    // (x-x0)^2+(y-y0)^2=r^2 ecuatia cercului verificata de punctele P1,P2,P3
    // 3 ecuatii si 3 necunoscute: x0, y0, r

    double a12, a13, b12, b13, c12, c13; // int ==> eroare !!!

    a12=2*(x1-x2); b12=2*(y1-y2); c12=x1*x1+y1*y1-x2*x2-y2*y2;
    a13=2*(x1-x3); b13=2*(y1-y3); c13=x1*x1+y1*y1-x3*x3-y3*y3;

    // sistemul devine: a12*x0+b12*y0=c12;
    //                  a13*x0+b13*y0=c13;
    if(a12*b13-a13*b12!=0)
    {
      x0=(c12*b13-c13*b12)/(a12*b13-a13*b12);
      y0=(a12*c13-a13*c12)/(a12*b13-a13*b12);
      r0=Math.sqrt((x1-x0)*(x1-x0)+(y1-y0)*(y1-y0));
    }
    else  // consider cercul de diametru [(minx,maxx),(miny,maxy)]
    { // punctele sunt coliniare !
      x0=(max(x1,x2,x3)+min(x1,x2,x3))/2;
      y0=(max(y1,y2,y3)+min(y1,y2,y3))/2;
      r0=d(x0,y0,x1,y1)/2;
    }
  }// cercCircumscris(...)

  static void cercDeDiametru(int x1,int y1,int x2,int y2)
  {
    x0=((double)x1+x2)/2;
    y0=((double)y1+y2)/2;
    r0=d(x1,y1,x2,y2)/2;
  }// cercDeDiametru(...)

  static int min(int a,int b) { if(a<b) return a; else return b; }
  static int max(int a,int b) { if(a>b) return a; else return b; }

  static int min(int a,int b,int c) { return min(min(a,b),min(a,c)); }
  static int max(int a,int b,int c) { return max(min(a,b),max(a,c)); }
```

```
  static double d(int x1, int y1, int x2, int y2)
  {
    double dx,dy;
    dx=x2-x1;
    dy=y2-y1;
    return Math.sqrt(dx*dx+dy*dy);
  }

  static double d(double x1, double y1, double x2, double y2)
  {
    double dx,dy;
    dx=x2-x1;
    dy=y2-y1;
    return Math.sqrt(dx*dx+dy*dy);
  }

  //interschimb punctele i si j
  static void swap(int i, int j)
  {
    int aux;
    aux=x[i]; x[i]=x[j]; x[j]=aux;
    aux=y[i]; y[i]=y[j]; y[j]=aux;
  }// swap(...)
}// class
```

18.9.3 Moşia lui Păcală - OJI2004 clasa a XI-a

Păcală a primit, aşa cum era învoiala, un petec de teren de pe moşia boierului. Terenul este împrejmuit complet cu segmente drepte de gard ce se sprijină la ambele capete de câte un par zdravăn. La o nouă prinsoare, Păcală iese iar în câştig şi primeşte dreptul să strămute nişte pari, unul câte unul, cum i-o fi voia, astfel încât să-şi extindă suprafaţa de teren. Dar învoiala prevede că fiecare par poate fi mutat în orice direcţie, dar nu pe o distanţă mai mare decât o valoare dată (scrisă pe fiecare par) şi fiecare segment de gard, fiind cam şuubred, poate fi rotit şi prelungit de la un singur capăt, celălalt rămânând nemişcat.

Cunoscnd poziţiile iniţiale ale parilor şi valoarea înscrisă pe fiecare par, se cere suprafaţa maximă cu care poate să-şi extindă Păcală proprietatea. Se ştie că parii sunt daţi într-o ordine oarecare, poziţiile lor iniţiale sunt date prin numere întregi de cel mult 3 cifre, distanţele pe care fiecare par poate fi deplasat sunt numere naturale strict pozitive şi figura formată de terenul iniţial este un poligon neconcav.

Date de intrare

Fişierul MOSIA.IN conţine $n+1$ linii cu următoarele valori:
n - numărul de pari
$x_1\ y_1\ d_1$ - coordonatele iniţiale şi distanţa pe care poate fi mutat parul 1
$x_2\ y_2\ d_2$ - coordonatele iniţiale şi distanţa pe care poate fi mutat parul 2
. . .
$x_n\ y_n\ d_n$ - coordonatele iniţiale şi distanţa pe care poate fi mutat parul n
Date de ieşire
În fişierul MOSIA.OUT se scrie un număr real cu 4 zecimale ce reprezintă suprafaţa maximă cu care se poate mări moşia.
Restricţii şi observaţii:
$3 < N \leq 200$ număr natural
$-1000 < x_i, y_i < 1000$ numere întregi
$0 < d_i \leq 20$ numere întregi
poligonul neconcav se defineşte ca un poligon convex cu unele vârfuri coliniare
poziţiile parilor sunt date într-o ordine oarecare
poligonul obţinut după mutarea parilor poate fi concav
poziţiile finale ale parilor nu sunt în mod obligatoriu numere naturale
Exemplu
Pentru fişierul de intrare

```
4
-3  0  2
3   0  3
0   6  2
0   -6 6
```

se va scrie în fişierul de ieşire valoarea 30.0000
Explicaţie: prin mutarea parilor 1 şi 2 cu câte 2 şi respectiv 3 unităţi, se obţine un teren având suprafaţa cu 30 de unităţi mai mare decât terenul iniţial.
Timp limită de executare: 1 sec./test

18.9.4 Partiţie - ONI2006 baraj

Ionică a primit de ziua lui de la tatăl său un joc format din piese de formă triunghiulară de dimensiuni diferite şi o suprafaţu a magnetică pe care acestea pot fi aşezate.

Pe suprafaţa magnetică este desenat un triunghi dreptunghic cu lungimile catetelor a, respectiv b şi un sistem de coordonate xOy cu originea în unghiul drept al triunghiului, semiaxa $[Ox$ pe cateta de lungime a, respectiv semiaxa $[Oy$ pe cateta de lungime b.

La un moment dat Ionică aşează pe tabla magnetică n piese, pentru care se cunosc coordonatele vârfurilor lor. Tatăl lui Ionică vrea să verifice dacă pe tablă piesele realizează o partiţie a triunghiului dreptunghic desenat, adică dacă sunt îndeplinite condiţiile:

• nu există piese suprapuse;

• piesele acoperă toată porţiunea desenată (în formă de triunghi dreptunghic);
• nu există porţiuni din piese în afara triunghiului desenat.

Cerinţă

Se cere să se verifice dacă piesele plasate pe tabla magnetică formează o partiţie a triunghiului desenat pe tabla magnetică.

Date de intrare

Fişierul de intrare **part.in** conţine pe prima linie un număr natural k, reprezentând numărul de seturi de date din fişier. Urmează k grupe de linii, câte o grupă pentru fiecare set de date. Grupa de linii corespunzătoare unui set este formată dintr-o linie cu numerele a, b, n separate între ele prin câte un spaţiu şi n linii cu câte şase numere întregi separate prin spaţii reprezentând coordonatele vârfurilor (abscisă ordonată) celor n piese, câte o piesă pe o linie.

Date de ieşire

În fişierul **part.out** se vor scrie k linii, câte o linie pentru fiecare set de date. Pe linia i ($i = 1, 2, ..., k$) se va scrie 1 dacă triunghiurile din setul de date i formează o partiţie a triunghiului desenat pe tabla magnetică sau 0 în caz contrar.

Restricţii şi precizări

• $1 \leq n \leq 150$
• $1 \leq k \leq 10$
• a, b sunt numere întregi din intervalul $[0, 31000]$
• Coordonatele vrfurilor pieselor sunt numere ntregi din intervalul [0, 31000].

Exemplu

part.in	part.out
2	1
20 10 4	0
0 5 0 10 10 5	
0 0 10 5 0 5	
0 0 10 0 10 5	
10 0 20 0 10 5	
20 10 2	
0 0 0 10 10 5	
0 0 20 0 20 10	

Timp maxim de execuţie: 0.3 secunde/test

Prelucrare în Java după rezolvarea în C a autorului problemei

```
import java.io.*;
class part
{
  static final int ON_EDGE=0;
  static final int INSIDE=1;
```

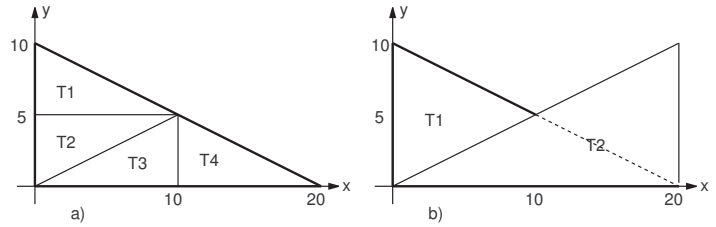


Figura 18.2: a) pentru setul 1 de date şi b) pentru setul 2 de date

```
static final int OUTSIDE=2;
static final int N_MAX=512;

static int N, A, B;
static int[][] X=new int[N_MAX][3];
static int[][] Y=new int[N_MAX][3];

static int sgn(int x)
{
  return x>0 ? 1 : (x<0 ? -1 : 0);
}

static int point_sign (int x1, int y1, int x2, int y2, int _x, int _y)
{
  int a, b, c;
  a=y2-y1;
  b=x1-x2;
  c=y1*x2-x1*y2;
  return sgn(a*_x+b*_y+c);
}

static int point_inside (int n, int x, int y)
{
  int i;
  int[] sgn=new int[3];
  for(i=0;i<3;i++)
    sgn[i]=point_sign(X[n][i],Y[n][i],X[n][(i+1)%3],Y[n][(i+1)%3],x,y);
  if(sgn[0]*sgn[1]<0 || sgn[0]*sgn[2]<0 || sgn[1]*sgn[2]<0) return OUTSIDE;
  if(sgn[0]==0 || sgn[1]==0 || sgn[2]==0)  return ON_EDGE;
  return INSIDE;
}
```

```
  static boolean segment_intersect(int x1,int y1,int x2,int y2,
                                   int x3,int y3,int x4,int y4)
  {
    int a1,b1,c1,a2,b2,c2;
    a1=y2-y1;  b1=x1-x2;  c1=y1*x2-x1*y2;
    a2=y4-y3;  b2=x3-x4;  c2=y3*x4-x3*y4;
    return sgn(a1*x3+b1*y3+c1)*sgn(a1*x4+b1*y4+c1)<0 &&
           sgn(a2*x1+b2*y1+c2)*sgn(a2*x2+b2*y2+c2)<0;
  }

  static boolean triangle_intersect (int n1, int n2)
  {
    int i,j,x,t1=0,t2=0;
    for(i=0;i<3;i++)
    {
      if((x=point_inside(n2,X[n1][i],Y[n1][i]))==ON_EDGE) t1++;
      if(x==INSIDE)  return true;
      if((x=point_inside(n1,X[n2][i],Y[n2][i]))==ON_EDGE) t2++;
      if(x==INSIDE) return true;
    }
    if(t1==3 || t2==3)  return true;
    for(i=0;i<3;i++)
      for(j=0;j<3;j++)
        if(segment_intersect(
             X[n1][i],Y[n1][i],X[n1][(i+1)%3],Y[n1][(i+1)%3],
             X[n2][j],Y[n2][j],X[n2][(j+1)%3],Y[n2][(j+1)%3]
        )) {  return true; }
    return false;
  }

  static int solve()
  {
    int i,j,area=0;
    for(i=0;i<N;i++)
    {
      for(j=0;j<3;j++)
        if(point_inside(N,X[i][j],Y[i][j])==OUTSIDE) return 0;

      area+=Math.abs((X[i][1]*Y[i][2]-X[i][2]*Y[i][1])-
                     (X[i][0]*Y[i][2]-X[i][2]*Y[i][0])+
                     (X[i][0]*Y[i][1]-X[i][1]*Y[i][0]));
    }

    if(area!=A*B)  return 0;
```

```
    for(i=0;i<N;i++)
      for(j=i+1;j<N;j++)
        if(triangle_intersect(i,j)) return 0;

    return 1;
  }

  public static void main(String[] args) throws IOException
  {
    int tests, i, j;
    StreamTokenizer st=new StreamTokenizer(
      new BufferedReader(new FileReader("part.in")));
    PrintWriter out=new PrintWriter(
      new BufferedWriter(new FileWriter("part.out")));

    st.nextToken(); tests=(int)st.nval;

    for(; tests-->0;)
    {
      st.nextToken(); A=(int)st.nval;
      st.nextToken(); B=(int)st.nval;
      st.nextToken(); N=(int)st.nval;

      for(i=0;i<N;i++)
        for(j=0;j<3;j++)
        {
          st.nextToken(); X[i][j]=(int)st.nval;
          st.nextToken(); Y[i][j]=(int)st.nval;
        }

      X[N][0]=0; Y[N][0]=0;
      X[N][1]=A; Y[N][1]=0;
      X[N][2]=0; Y[N][2]=B;
      out.println(solve());
    }
    out.close();
  }// main(...)
}// class
```

18.9.5 Triunghi - ONI2007 cls 9

În comuna **Triunghi** din România sunt n ţărani codificaţi prin numerele

$1, 2, ..., n$. După anul 1990 a început retrocedarea suprafeţelor de pământ deţinute înainte de colectivizare. Fiecare ţăran are un document prin care dovedeşte că este proprietar pe o singură suprafaţă de teren de formă triunghiulară. Din păcate, documentele dau bătaie de cap primarului (care se ocupă de retrocedarea suprafeţelor de pământ), pentru că sunt porţiuni din suprafeţele de pământ care se regăsesc pe mai multe documente.

În această comună există o fântână cu apă, fiind posibil ca ea să fie revendicată de mai mulţi ţărani. O suprafaţă de pământ este dată prin coordonatele celor trei colţuri, iar fântâna este considerată punctiformă şi dată prin coordonatele punctului.

Cerinţă

Să se scrie un program care să determine:

a) Codurile ţăranilor care au documente cu suprafeţe de pământ ce conţin în interior sau pe frontieră fântâna.

b) Codul ţăranului ce deţine un document cu suprafaţa de teren, care include toate celelalte suprafeţe.

Date de intrare

Fişierul de intrare **triunghi.in** are pe prima linie numărul n de ţărani, pe următoarele n linii câte 6 valori numere întregi separate prin câte un spaţiu, în formatul: $x1\ y1\ x2\ y2\ x3\ y3$, ce reprezintă coordonatele celor trei colţuri ale suprafeţei triunghiulare deţinute de un ţăran ($x1$, $x2$, $x3$ abscise, iar $y1$, $y2$, $y3$ ordonate). Pe linia $i+1$ se află coordonatele colţurilor suprafeţei de teren triunghiulare deţinute de ţăranul i, $i = 1, 2, ..., n$. Ultima linie a fişierului (linia $n+2$) va conţine coordonatele fântânii în formatul $x\ y$, cu un spaţiu între ele (x abscisă, iar y ordonată).

Date de ieşire

Fişierul de ieşire **triunghi.out** va conţine pe prima linie răspunsul de la punctul *a*), adică: numărul de ţărani care îndeplinesc condiţia din cerinţă şi apoi codurile lor (în ordine crescătoare), cu un spaţiu între ele. Dacă nu există ţărani cu condiţia din cerinţă, pe prima linie se va scrie cifra 0. Pe linia a doua se va scrie răspunsul de la punctul *b*), adică: codul ţăranului cu proprietatea cerută, sau cifra 0, dacă nu există un astfel de ţăran.

Restricţii şi precizări

- $2 \leq n \leq 65$
- coordonatele colţurilor suprafeţelor de pământ şi ale fântânii sunt numere întregi din intervalul $[-3000, 3000]$
- cele trei colţuri ale fiecărei suprafeţe de pământ sunt distincte şi necoliniare
- nu există doi ţărani care să deţină aceeaşi suprafaţă de pământ
- nu se acordă punctaje parţiale.

Exemplu

secv.in	secv.out
3 10 0 0 10 10 10 0 100 100 0 -100 0 0 0 10 0 0 10 10 5	2 1 2 2

Explicaţie:

La punctul *a*), sunt doi ţărani care deţin suprafeţe de pământ ce au în interior sau pe frontieră fântâna, cu codurile 1 şi 2.

La punctul *b*), ţăranul cu codul 2 deţine o suprafaţă de teren care include, suprafeţele de pământ deţinute de ceilalţi ţărani (cu codurile 1 şi 3).

Timp maxim de execuţie/test: 0.1 secunde

Indicaţii de rezolvare - descriere soluţie

Descrierea soluţiei (Prof. Doru Popescu Anastasiu)

Notăm cu $T_1, T_2, ..., T_n$ triunghiurile corespunzătoare suprafeţelor şi cu I punctul unde se găseşte fântâna.

$$T_i = A_iB_iC_i, \;\; i = 1, 2, ..., n.$$

a)

$nr = 0$

Pentru $i = 1, ..., n$ verificăm dacă I este interior sau pe frontiera lui T_i, în caz afirmativ $nr = nr + 1$ şi $sol[nr] = i$. Afişăm nr şi vectorul sol.

Pentru a verifica dacă I este interior sau pe frontiera unui triunghi T_i este suficient să verificăm dacă:

$$aria(A_iB_iC_i) = aria(IA_iB_i) + aria(IA_iC_i) + aria(IB_iC_i)$$

O altă variantă ar fi să folosim poziţia unui punct faţă de o dreaptă.

b)

Dacă există un asemenea triunghi atunci el este de arie maximă. Astfel determinăm triunghiul p de arie maximă. Pentru acest triunghi verificăm dacă toate celelalte $n - 1$ triunghiuri sunt interioare sau pe frontiera lui T_p (adică dacă au toate vârfurile în interiorul sau pe frontiera lui T_p). În caz afirmativ se afişează p, altfel 0.

Codul sursă

```
import java.io.*;
class Pereti
{
```

```
static int n,x0,y0;
static int smax=-1,imax=-1,nr=0;
static int[] x1=new int[66];
static int[] y1=new int[66];
static int[] x2=new int[66];
static int[] y2=new int[66];
static int[] x3=new int[66];
static int[] y3=new int[66];
static int[] sol=new int[66];

public static void main(String[] args) throws IOException
{
  int i,j,s,s1,s2,s3;
  boolean ok;

  StreamTokenizer st=new StreamTokenizer(
    new BufferedReader(new FileReader("14-triunghi.in")));
  PrintWriter out=new PrintWriter(
    new BufferedWriter(new FileWriter("triunghi.out")));
  st.nextToken(); n=(int)st.nval;
  for(i=1;i<=n;i++)
  {
    st.nextToken(); x1[i]=(int)st.nval;
    st.nextToken(); y1[i]=(int)st.nval;
    st.nextToken(); x2[i]=(int)st.nval;
    st.nextToken(); y2[i]=(int)st.nval;
    st.nextToken(); x3[i]=(int)st.nval;
    st.nextToken(); y3[i]=(int)st.nval;
  }
  st.nextToken(); x0=(int)st.nval;
  st.nextToken(); y0=(int)st.nval;
  for(i=1;i<=n;i++)
  {
    s=aria(x1[i],y1[i],x2[i],y2[i],x3[i],y3[i]);
    if(s>smax) {smax=s; imax=i;}
    s1=aria(x1[i],y1[i],x2[i],y2[i],x0,y0);
    s2=aria(x2[i],y2[i],x3[i],y3[i],x0,y0);
    s3=aria(x1[i],y1[i],x3[i],y3[i],x0,y0);
    if(s==s1+s2+s3) {nr++; sol[nr]=i;}
    //System.out.println("i = "+i+" --> "+s+" "+s1+" "+s2+" "+s3);
  }
  if(nr>0)
  {
    out.print(nr+" ");
```

```
      for(i=1;i<=nr;i++)
        if(i!=nr) out.print(sol[i]+" "); else out.println(sol[i]);
    }
    else out.println(0);

    //System.out.println("imax = "+imax);
    ok=true;
    for(i=1;i<=n;i++)
    {
      if(i==imax) continue;

      s1=aria(x1[imax],y1[imax],x2[imax],y2[imax],x1[i],y1[i]);
      s2=aria(x2[imax],y2[imax],x3[imax],y3[imax],x1[i],y1[i]);
      s3=aria(x1[imax],y1[imax],x3[imax],y3[imax],x1[i],y1[i]);
      if(smax!=s1+s2+s3) { ok=false; break; }

      s1=aria(x1[imax],y1[imax],x2[imax],y2[imax],x2[i],y2[i]);
      s2=aria(x2[imax],y2[imax],x3[imax],y3[imax],x2[i],y2[i]);
      s3=aria(x1[imax],y1[imax],x3[imax],y3[imax],x2[i],y2[i]);
      if(smax!=s1+s2+s3) { ok=false; break; }

      s1=aria(x1[imax],y1[imax],x2[imax],y2[imax],x3[i],y3[i]);
      s2=aria(x2[imax],y2[imax],x3[imax],y3[imax],x3[i],y3[i]);
      s3=aria(x1[imax],y1[imax],x3[imax],y3[imax],x3[i],y3[i]);
      if(smax!=s1+s2+s3) { ok=false; break; }
    }
    if(ok) out.println(imax); else out.println(0);
    out.close();
  }// main(...)

  static int aria(int x1, int y1, int x2, int y2, int x3, int y3) // dubla ...
  {
    int s=x1*y2+x2*y3+x3*y1-y1*x2-y2*x3-y3*x1;
    if(s<0) s=-s;
    return s;
  }
}// class
```

Capitolul 19

Teoria jocurilor

19.1 Jocul NIM

19.1.1 Prezentare generală

19.1.2 Exemple

Capitolul 20

Alţi algoritmi

20.1 Secvenţă de sumă maximă

20.1.1 Prezentare generală

20.1.2 Exemple

20.2 Algoritmul Belmann-Ford

Pentru grafuri cu costuri negative (dar fara cicluri negative!) se poate folosi algoritmul Bellman-Ford.

20.2.1 Algoritmul Belmann-Ford pentru grafuri neorientate

```
// drum scurt in graf neorientat cu costuri negative (dar fara ciclu negativ!)
// Algoritm: 1. init
//           2. repeta de n-1 ori
//                 pentru fiecare arc (u,v)
//                    relax(u,v)
//           3. OK=true
```

```
//           4. pentru fiecare muchie (u,v)
//                 daca d[v]>d[u]+w[u][v]
//                    OK=false
//           5. return OK

import java.io.*;
class BellmanFord
{
static final int oo=0x7fffffff; // infinit
static int n,m;                 // varfuri, muchii

static int[][] w;               // matricea costurilor
static int[] d;                 // d[u]=dist(nods,u)
static int[] p;                 // p[u]=predecesorul nodului u

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    int nods=1, cost;      // nod sursa, costul arcului
    int u,v;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("BellmanFordNeorientat.in")));

    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    w=new int[n+1][n+1];
    d=new int[n+1];
    p=new int[n+1];

    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
        w[i][j]=oo;// initializare !

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost=(int)st.nval;
      w[i][j]=cost;
      w[j][i]=cost; // numai pentru graf neorientat
    }

    init(nods);
```

```
  for(k=1;k<=n-1;k++)       // de n-1 ori !!!
    for(u=1;u<n;u++)        // vectorii muchiilor erau mai buni !
      for(v=u+1;v<=n;v++)   // lista de adiacenta era mai buna !
        if(w[u][v]<oo)      // (u,v)=muchie si u<v
          relax(u,v);

  boolean cicluNegativ=false;
  for(u=1;u<n;u++)
    for(v=u+1;v<=n;v++)
      if(w[u][v]<oo)          // (u,v)=muchie
        if(d[u]<oo)           // atentie !!! oo+ceva=???
          if(d[v]>d[u]+w[u][v])
          {
            cicluNegativ=true;
            break;
          }

  if(!cicluNegativ)
    for(k=1;k<=n;k++)
    {
      System.out.print(nods+"-->"+k+" dist="+d[k]+" drum: ");
      drum(k);
      System.out.println();
    }
}//main

static void init(int s)
{
  int u;
  for(u=1;u<=n;u++) { d[u]=oo;  p[u]=-1;  }
  d[s]=0;
}// init(...)

static void relax(int u,int v) // (u,v)=arc(u-->v)
{
  if(d[u]<oo) // oo+ceva ==> ???
  if(d[u]+w[u][v]<d[v])
  {
    d[v]=d[u]+w[u][v];
    p[v]=u;
  }
}
```

```
  static void drum(int k) // s --> ... --> k
  {
    if(p[k]!=-1) drum(p[k]);
    System.out.print(k+" ");
  }

  static void afisv(int[] x)
  {
    int i;
    for(i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}//class

/*
6 7
1 2 -3      1-->1 dist=0 drum: 1
1 3 1       1-->2 dist=-3 drum: 1 2
2 3 2       1-->3 dist=-1 drum: 1 2 3
3 4 1       1-->4 dist=0 drum: 1 2 3 4
4 5 1       1-->5 dist=1 drum: 1 2 3 4 5
5 6 -3      1-->6 dist=-2 drum: 1 2 3 4 5 6
4 6 2
*/
```

20.2.2 Alg Belmann-Ford pentru grafuri orientate

```
// drumuri scurte in graf orientat cu costuri negative (dar fara ciclu negativ!)
// Dijkstra nu functioneaza daca apar costuri negative !
// Algoritm:  1. init
//            2. repeta de n-1 ori
//                  pentru fiecare arc (u,v)
//                     relax(u,v)
//            3. OK=true
//            4. pentru fiecare arc (u,v)
//                  daca d[v]>d[u]+w[u][v]
//                     OK=false
//            5. return OK

import java.io.*;
class BellmanFord
{
  static final int oo=0x7fffffff;
  static int n,m; // varfuri, muchii
```

```
static int[][] w; // matricea costurilor
static int[] d; // d[u]=dist(nods,u)
static int[] p; // p[u]=predecesorul nodului u

public static void main(String[] args) throws IOException
{
  int i,j,k;
  int nods=1, cost; // nod sursa, costul arcului
  int u,v;
  StreamTokenizer st=new StreamTokenizer(
                     new BufferedReader(new FileReader("BellmanFordNeorientat.in")));
  st.nextToken();  n=(int)st.nval;
  st.nextToken(); m=(int)st.nval;

  w=new int[n+1][n+1];
  d=new int[n+1];
  p=new int[n+1];
  for(i=1;i<=n;i++) for(j=1;j<=n;j++) w[i][j]=oo; // initializare !
  for(k=1;k<=m;k++)
  {
    st.nextToken(); i=(int)st.nval;
    st.nextToken(); j=(int)st.nval;
    st.nextToken(); cost=(int)st.nval;
    w[i][j]=cost;
  }

  init(nods);
  for(k=1;k<=n-1;k++)     // de n-1 ori !!!
    for(u=1;u<=n;u++)     // vectorii arcelor erau mai buni !
      for(v=1;v<=n;v++)   // lista de adiacenta era mai buna !
        if(w[u][v]<oo)    // (u,v)=arc
          relax(u,v);

  boolean cicluNegativ=false;
  for(u=1;u<=n;u++)
    for(v=1;v<=n;v++)
      if(w[u][v]<oo)       // (u,v)=arc
        if(d[u]<oo)        // atentie !!! oo+ceva=???
          if(d[v]>d[u]+w[u][v])
          {
            cicluNegativ=true;
            break;
          }
  System.out.println(cicluNegativ);
```

```
    if(!cicluNegativ)
      for(k=1;k<=n;k++)
      {
        System.out.print(nods+"-->"+k+" dist="+d[k]+" drum: ");
        if(d[k]<oo) drum(k); else System.out.print("Nu exista drum!");
        System.out.println();
      }
  }//main

  static void init(int s)
  {
    int u;
    for(u=1;u<=n;u++) { d[u]=oo; p[u]=-1; }
    d[s]=0;
  }// init()

  static void relax(int u,int v) // (u,v)=arc(u-->v)
  {
    if(d[u]<oo)                    // oo+ceva ==> ???
      if(d[u]+w[u][v]<d[v])
      {
        d[v]=d[u]+w[u][v];
        p[v]=u;
      }
  }// relax(...)

  static void drum(int k) // s --> ... --> k
  {
    if(p[k]!=-1) drum(p[k]);
    System.out.print(k+" ");
  }// drum(...)

  static void afisv(int[] x)
  {
    int i;
    for(i=1;i<=n;i++) System.out.print(x[i]+" ");
    System.out.println();
  }
}//class

/*
6 8             false
1 2 -3          1-->1 dist=0 drum: 1
1 3 1           1-->2 dist=-4 drum: 1 3 2
```

```
2 3 6           1-->3 dist=1 drum: 1 3
3 4 1           1-->4 dist=2 drum: 1 3 4
5 4 1           1-->5 dist=2147483647 drum: Nu exista drum!
5 6 -3          1-->6 dist=4 drum: 1 3 4 6
4 6 2
3 2 -5
*/
```

20.2.3 Alg Belmann-Ford pentru grafuri orientate aciclice

```
// Cele mai scurte drumuri in digraf (graf orientat ACICLIC)
// Dijkstra nu functioneaza daca apar costuri negative !
// Algoritm:  1. sortare topologica O(n+m)
//            2. init(G,w,s)
//            3. pentru toate nodurile u in ordine topologica
//                  pentru toate nodurile v adiacente lui u
//                     relax(u,v)
// OBS: O(n+m)

import java.io.*;
class BellmanFordDAG
{
  static final int oo=0x7fffffff;
  static final int WHITE=0, BLACK=1; // color[u]=BLACK ==> u in lista
  static int n,m,t,pozl;   // varfuri, muchii, time, pozitie in lista
  static int[] color;      // culoare
  static int[] lista;      // lista
  static int[] gi;         // grad interior
  static int[][] w;        // matricea costurilor
  static int[] d;          // d[u]=dist(nods,u)
  static int[] p;          // p[u]=predecesorul nodului u

  public static void main(String[] args) throws IOException
  {
    int i,j,k;
    int nods=1, cost; // nod sursa, costul arcului
    int u,v;
    StreamTokenizer st=new StreamTokenizer(
                       new BufferedReader(new FileReader("BellmanFordDAG.in")));
    st.nextToken(); n=(int)st.nval;
    st.nextToken(); m=(int)st.nval;

    w=new int[n+1][n+1];
    color=new int[n+1];
```

```
    lista=new int[n+1];
    gi=new int[n+1];
    d=new int[n+1];
    p=new int[n+1];

    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++) w[i][j]=oo; // initializare !

    for(k=1;k<=m;k++)
    {
      st.nextToken(); i=(int)st.nval;
      st.nextToken(); j=(int)st.nval;
      st.nextToken(); cost=(int)st.nval;
      w[i][j]=cost;
      gi[j]++;
    }

    topoSort();
    System.out.print("Lista    : "); afisv(lista);

    init(nods);
    for(i=1;i<=n;i++)
    {
      u=lista[i];
      for(v=1;v<=n;v++)
        if(w[u][v]<oo)    // lista de adiacenta era mai buna !
          relax(u,v);
    }

    System.out.print("Distante : ");
    afisv(d);

    for(k=1;k<=n;k++)
    {
      if(d[k]<oo) System.out.print(k+" : "+d[k]+" ... ");
      else        System.out.print(k+": oo ... ");
      drum(k);
      System.out.println();
    }
  }//main

  static void init(int s)
  {
    int u;
```

```
  for(u=1;u<=n;u++)
  {
    d[u]=oo;
    p[u]=-1;
  }
  d[s]=0;
}// init(...)

static void relax(int u,int v) // (u,v)=arc(u-->v)
{
  if(d[u]<oo)                    // oo+ceva ==> ???
    if(d[u]+w[u][v]<d[v]) { d[v]=d[u]+w[u][v]; p[v]=u; }
}// relax(...)

static void drum(int k) // s --> ... --> k
{
  if(p[k]!=-1) drum(p[k]);
  if(d[k]<oo) System.out.print(k+" ");
}

static void topoSort()
{
  int u,i,k,pozl;
  for(i=1;i<=n;i++) // oricum era initializat implicit, dar ... !!!
    color[i]=WHITE;
  pozl=1;
  for(k=1;k<=n;k++) // pun cate un nod in lista
  {
    u=nodgi0();
    color[u]=BLACK;
    micsorezGrade(u);
    lista[pozl++]=u;
  }
}// topoSort()

static int nodgi0() // nod cu gradul interior zero
{
  int v,nod=-1;
  for(v=1;v<=n;v++) // coada cu prioritati (heap) este mai buna !!!
    if(color[v]==WHITE)
      if(gi[v]==0) {nod=v; break;}
  return nod;
}// nodgi0()
```

```
  static void micsorezGrade(int u)
  {
    int v;
    for(v=1;v<=n;v++) // lista de adiacenta este mai buna !!!
    if(color[v]==WHITE)
      if(w[u][v]<oo) gi[v]--;
  }// micsorezGrade(...)

  static void afisv(int[] x)
  {
    int i;
    for(i=1;i<=n;i++)
    if(x[i]<oo) System.out.print(x[i]+" "); else System.out.print("oo ");
    System.out.println();
  }// afisv(...)
}//class

/*
6 7           Lista    : 1 3 2 5 4 6
1 2 -3        Distante : 0 -4 1 2 oo 4
1 3 1         1 : 0 ... 1
3 4 1         2 : -4 ... 1 3 2
5 4 1         3 : 1 ... 1 3
5 6 -3        4 : 2 ... 1 3 4
4 6 2         5: oo ...
3 2 -5        6 : 4 ... 1 3 4 6
*/
```

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