

Emanuela Cerchez (n. 12.04.1968, la Iași) este absolventă a Facultății de Matematică, secția Informatică (1990), și a Seminarului pedagogic postuniversitar (1997), profesoră de informatică (grad didactic I), membră în Comisia Națională de Informatică. Autoarea a mai publicat la Editura Polirom : *Internet. Manual pentru liceu* (2000, avizat MEN), *Informatica. Manual pentru clasa a X-a* (coautor Marinel-Paul Șerban, 2000, avizat MEN), *PC. Pas cu pas* (coautor Marinel-Paul Șerban, 2001), *Informatica pentru gimnaziu* (coautor Marinel-Paul Șerban, 2002, avizat MEC), *Informatica. Culegere de probleme pentru liceu* (2002), *Programarea în limbajul C/C++ pentru liceu* (vol. I, coautor Marinel-Paul Șerban, 2005), *Programarea în limbajul C/C++ pentru liceu. Metode și tehnici de programare* (vol. II, coautor Marinel-Paul Șerban, 2005).

Marinel-Paul Șerban (n. 29.06.1950, la Arad) este absolvent al Facultății de Matematică-Mecanică, Universitatea din Timișoara (1973); specializare postuniversitară în informatică (1974), profesor de informatică (grad didactic I), membru în Comisia Națională de Informatică. De același autor, la Editura Polirom au apărut : *Informatica. Manual pentru clasa a X-a* (coautoare Emanuela Cerchez, 2000, avizat MEN), *PC. Pas cu pas* (coautoare Emanuela Cerchez, 2001), *Informatica pentru gimnaziu* (coautoare Emanuela Cerchez, 2002, avizat MEC), *Programarea în limbajul C/C++ pentru liceu* (vol. I, coautoare Emanuela Cerchez, 2005), *Programarea în limbajul C/C++ pentru liceu. Metode și tehnici de programare* (vol. II, coautoare Emanuela Cerchez, 2005).

Autorii au o bogată experiență în pregătirea de performanță a elevilor : susțin cursuri la Centrul de Pregătire a Tinerilor Capabili de Performanță din Iași, propun probleme pentru olimpiadele și concursurile naționale și județene de informatică, susțin activitatea de pregătire a lotului național de informatică, precum și programul de pregătire de performanță în informatică *campion*.

© 2006 by Editura POLIROM

[www.polirom.ro](http://www.polirom.ro)

Editura POLIROM

Iași, B-dul Carol I nr. 4, P.O. BOX 266, 700506

București, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, O.P. 37; P.O. BOX 1-728, 030174

Descrierea CIP a Bibliotecii Naționale a României :

CERCHEZ, EMANUELA

Programarea în limbajul C/C++ pentru liceu / Emanuela Cerchez, Marinel Șerban. – Iași :  
Polirom, 2006

3 vol.

ISBN (10) 973-46-0109-1

Vol. 3. – 2006. – ISBN (10) : 973-46-0438-4 ; ISBN (13) : 978-973-46-0438-8

I. Șerban, Marinel

004.42(075.35)

004.43 C(075.35)

004.43 C++(075.35)

Printed in ROMANIA

**Emanuela Cerchez, Marinel Șerban**

# **PROGRAMAREA ÎN LIMBAJUL C/C++ PENTRU LICEU**



- Teoria grafurilor
- Structuri de date dinamice (liniare și arborescente)
- Pattern-matching
- Hashing
- Geometrie computațională

POLIROM  
2006

## Cuprins

1. Teoria grafurilor.....	7
1.1. Noțiuni introductive .....	7
1.2. Reprezentarea grafurilor în memorie.....	24
1.3. Grafuri ponderate. Reprezentare .....	36
1.4. Închiderea tranzitivă a unui graf .....	37
1.5. Parcursarea grafurilor .....	38
1.6. Conexitate.....	46
1.7. Tare-conexitate.....	51
1.8. Arbori .....	54
1.9. Arbori parțiali .....	58
1.10. Arbori parțiali de cost minim .....	62
1.11. Biconexitate .....	74
1.12. Descompunere pe niveluri a unui graf fără circuite .....	78
1.13. Sortare topologică .....	81
1.14. Grafuri hamiltoniene.....	83
1.15. Grafuri euleriene .....	84
1.16. Drumuri minime în graf.....	87
1.17. Rețele de transport .....	95
1.18. Cuplaj maximal în graf bipartit .....	101
1.19. Aplicații rezolvate.....	103
1.20. Aplicații propuse .....	135
2. Liste înlățuite .....	149
2.1. Liste simplu înlățuite .....	149
2.2. Liste simplu înlățuite circulare .....	160
2.3. Liste dublu înlățuite .....	163
3. Structuri de date arborescente .....	167
3.1. Terminologie .....	167
3.2. Arbori binari .....	168
3.3. Reprezentarea arborilor cu rădăcină .....	172
3.4. Crearea unui arbore binar .....	175
3.5. Parcursarea arborilor binari .....	177

3.6. Determinarea înălțimii unui arbore .....	179
3.7. Crearea unui arbore binar pe baza parcurgerilor în preordine și inordine.....	179
3.8. <i>Heap-uri</i> .....	182
3.9. Arbori binari de căutare .....	198
3.10. Reprezentarea mulțimilor disjuncte .....	208
3.11. Arbori de compresie Huffman .....	212
3.12. Arbori asociati expresiilor aritmetice .....	216
3.13. Aplicații rezolvate.....	219
3.14. Aplicații propuse .....	226
<b>4. Pattern-matching .....</b>	<b>233</b>
4.1. Introducere.....	233
4.2. Algoritmul elementar (naiv) .....	233
4.3. Algoritmul Knuth-Morris-Pratt (KMP) .....	234
4.4. Aplicație. Desert.....	236
4.5. Probleme propuse.....	238
<b>5. Hashing .....</b>	<b>240</b>
5.1. Introducere.....	240
5.2. Funcții hash .....	240
5.3. Rezolvarea coliziunilor .....	242
5.4. Hashing perfect .....	244
5.5. Algoritmul Rabin Karp .....	245
5.6. Probleme propuse.....	247
<b>6. Geometrie computațională .....</b>	<b>251</b>
6.1. Introducere.....	251
6.2. Puncte și drepte .....	251
6.3. Poligoane.....	255
6.4. Geometria dreptunghiului .....	266
6.5. Aplicații .....	269
6.6. Probleme propuse.....	278
<b>7. Soluții și indicații.....</b>	<b>284</b>
<b>Bibliografie .....</b>	<b>293</b>

## 1. Teoria grafurilor

Numerose situații din viața cotidiană pot fi modelate cu ajutorul teoriei grafurilor. Numeroase probleme practice, cu aplicații în fizică, economie, chimie, sociologie etc., își găsesc soluția utilizând algoritmi din teoria grafurilor.

Există două mari categorii de grafuri: grafuri orientate și grafuri neorientate. Vom încerca să le prezentăm în paralel datorită numeroaselor similarități care există între noțiunile fundamentale, reprezentările în memorie și algoritmii corespunzători celor două categorii de grafuri.

### 1.1. Noțiuni introductive

Un *graf* (orientat sau neorientat) este o pereche ordonată de mulțimi  $G = (V, E)$ . Mulțimea  $V$  este o mulțime nevidă și finită de elemente denumite *vârfurile* grafului. Mulțimea  $E$  este o mulțime de perechi de vârfuri din graf. În cazul grafurilor orientate, perechile de vârfuri din mulțimea  $E$  sunt ordonate și sunt denumite *arce*; în cazul grafurilor neorientate, perechile de vârfuri din mulțimea  $E$  sunt neordonate și sunt denumite *muchii*.

Perechea ordonată formată din vârfurile  $x$  și  $y$  se notează  $(x, y)$ ; vârful  $x$  se numește *extremitate initială* a arcului  $(x, y)$ , iar vârful  $y$  se numește *extremitate finală* a arcului  $(x, y)$ .

Perechea neordonată formată din vârfurile  $x$  și  $y$  se notează  $[x, y]$ ; vârfurile  $x$  și  $y$  se numesc *extremitățile* muchiei  $[x, y]$ .

Dacă există un arc sau o muchie cu extremitățile  $x$  și  $y$ , atunci vârfurile  $x$  și  $y$  sunt *adiacente*; fiecare extremitate a unei muchii/unui arc este considerată *incidentă* cu muchia/arcul respectiv.

Vom considera că extremitățile unei muchii, respectiv ale unui arc sunt distincte (adică graful nu conține bucle).

#### Observații

1. Cu ajutorul unui graf neorientat putem modela o relație simetrică între elementele unei mulțimi, în timp ce cu ajutorul unui graf orientat modelăm o relație care nu este simetrică.

2. Între oricare două vârfuri ale unui graf poate exista cel mult o muchie/arc. Dacă între două vârfuri există mai multe muchii/arce atunci structura se numește *multigraf*.
3. În practică, informațiile asociate unui vârf din graf pot fi oricără de complexe, dar, pentru a simplifica, noi vom considera că vârfurile grafului sunt etichetate cu numere naturale de la 1 la n (unde cu n vom nota numărul de vârfuri din graf). Această numărare nu este o restrângere a generalității (numărul vârfului poate fi considerat, de exemplu, poziția pe care sunt memorate într-un vector informațiile asociate vârfului).
4. În unele lucrări de specialitate, un vârf al grafului este denumit *nod*.

#### *Exemple*

1. Să considerăm o clasă formată din n elevi. Unii dintre elevii clasei sunt prieteni, relația de prietenie fiind evident simetrică (adică, dacă Gigel este prieten cu Ionel, atunci și Ionel este prieten cu Gigel). Putem modela relațiile de prietenie din clasă cu ajutorul unui graf neorientat în care mulțimea vârfurilor are n elemente (elevii clasei), iar mulțimea muchiilor este formată din toate perechile de elevi, cu proprietatea că elevii care formează perechea sunt prieteni.
2. Să considerăm aceeași clasă de elevi. Doamna profesoară de română încurajează schimburile de cărți între elevii clasei și chiar i-a sfătuin să completeze într-un caiet special împrumuturile (ce carte a fost împrumutată, de către cine și cui). Această situație poate fi modelată cu ajutorul unui graf orientat în care mulțimea vârfurilor corespunde elevilor clasei, iar mulțimea arcelor corespunde împrumuturilor efectuate (există arc de la vârful x la vârful y dacă elevul corespunzător vârfului x a împrumutat o carte de la elevul corespunzător vârfului y). Graful este orientat deoarece relația definită nu este simetrică (elevul x poate împrumuta cărți de la elevul y, fără ca elevul y să împrumute cărți de la elevul x). Graful astfel definit modelează doar relația de împrumut existentă între elevii clasei (a împrumutat sau nu elevul x cărți de la elevul y), dar nu și împrumuturile efective. Dacă dorim să modelăm fiecare împrumut efectuat, vom construi un multigraf orientat în care vom avea câte un arc de la x la y pentru fiecare carte împrumutată de elevul x de la elevul y).

#### *Exerciții propuse*

1. Să analizăm pe hartă rețeaua stradală a orașului în care locuim. Observăm că unele străzi au sens unic, în timp ce alte străzi au două sensuri de circulație. De asemenea, orice stradă are două capete. Capetele de străzi pot fi intersecții sau fundături. Modelați harta cu ajutorul unui graf, explicând dacă graful este orientat sau neorientat, care este mulțimea vârfurilor grafului și, de asemenea, care este mulțimea muchiilor/arcelor grafului.
2. Să analizăm harta lumii. Cum se pot modela relațiile de vecinătate dintre țări cu ajutorul unui graf?
3. Dați exemple de două situații din viața cotidiană care pot fi modelate cu ajutorul unui graf orientat și de două situații care pot fi modelate cu ajutorul unui graf neorientat. Explicați modul în care construjiți mulțimea vârfurilor, precum și mulțimea arcelor/muchiilor grafului.

#### *Reprezentarea vizuală a grafurilor*

Pentru o mai bună înțelegere a noțiunii de graf se utilizează o reprezentare vizuală descrisă astfel:

- fiecărui vârf din graf îi corespunde un punct în plan, în dreptul căruia este specificat numărul vârfului;
- dacă graful este orientat, vom reprezenta fiecare arc ca o săgeată dinspre extremitatea inițială către extremitatea finală a arcului;
- dacă graful este neorientat, vom reprezenta fiecare muchie ca o linie (dreaptă sau curbă) care unește cele două extremități ale muchiei.

Uneori, pentru o mai mare lizibilitate, un vârf se reprezintă ca un cerc sau un pătrat în interiorul căruia se specifică numărul vârfului, ori un disc lângă care se specifică numărul vârfului. Pot exista și alte variante de vizualizare a unui vârf; noi vom utiliza în continuare forma din exemplele următoare.

#### *Exemple*

1. Să considerăm următorul graf orientat  $G=(V, E)$ , unde  $V=\{1, 2, 3, 4, 5\}$ , iar  $E=\{(1, 2), (1, 4), (2, 4), (3, 2), (2, 3)\}$ . Acest graf poate fi reprezentat astfel:

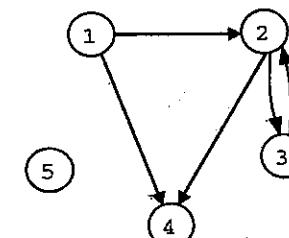


Figura 1

2. Graful neorientat  $G=(V, E)$ , unde  $V=\{1, 2, 3, 4, 5\}$ , iar  $E=\{[1, 2], [1, 4], [2, 4], [3, 2]\}$ , poate fi reprezentat astfel:

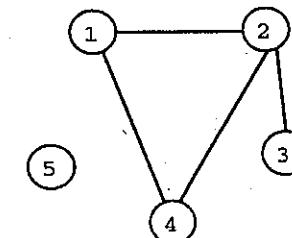


Figura 2

### Exerciții propuse

- Reprezentați vizual graful neorientat cu mulțimea vârfurilor  $\{1, 2, 3, 4, 5, 6\}$  și mulțimea muchiilor  $\{(1,2), (1,3), (1,4), (1,5), (1,6), (2,3), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6), (4,5), (4,6), (5,6)\}$ .
- Reprezentați vizual graful orientat cu mulțimea vârfurilor  $\{1, 2, 3, 4, 5, 6\}$  și mulțimea arcelor  $\{(1,2), (1,3), (1,4), (1,5), (5,1), (6,1), (3,6)\}$ .

### Gradul unui vârf

Fie  $G=(V, E)$  un graf neorientat. Se numește *grad* al unui vârf din graf numărul de muchii incidente cu vârful respectiv. Gradul vârfului  $x$  se notează  $d(x)$ .

De exemplu, pentru graful neorientat ilustrat în figura 2:

$$d(1)=2, d(2)=3, d(3)=1, d(4)=2, d(5)=0.$$

Se numește *vârf izolat* un vârf care are gradul 0. Se numește *vârf terminal* un vârf cu gradul 1.

De exemplu, în graful neorientat din figura 2, vârful 5 este vârf izolat, iar 3 este vârf terminal.

Fie  $G=(V, E)$  un graf orientat și  $x$  un vârf din graf. *Gradul exterior* al vârfului  $x$  se notează  $d^+(x)$  și este egal cu numărul de arce care au ca extremitate inițială pe  $x$ . *Gradul interior* al vârfului  $x$  se notează  $d^-(x)$  și este egal cu numărul de arce care au ca extremitate finală pe  $x$ .

De exemplu, pentru graful orientat ilustrat în figura 1:

$$\begin{aligned} d^+(1) &= 2, d^+(2) = 2, d^+(3) = 1, d^+(4) = 0, d^+(5) = 0 \\ d^-(1) &= 0, d^-(2) = 2, d^-(3) = 1, d^-(4) = 2, d^-(5) = 0. \end{aligned}$$

### Teorema 1

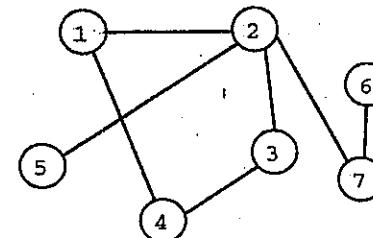
- Suma gradelor vârfurilor unui graf neorientat este egală cu dublul numărului de muchii din graf.
- Suma gradelor interioare ale vârfurilor unui graf orientat este egală cu suma gradelor exterioare ale vârfurilor grafului și egală cu numărul de arce din graf.

### Demonstrație

Într-un graf neorientat, fiecare muchie din graf contribuie cu o unitate la gradul fiecărei dintre extremitățile sale, deci în total la suma tuturor gradelor vârfurilor, fiecare muchie contribuie cu două unități. Într-un graf orientat, fiecare arc contribuie cu o unitate la suma gradelor interioare ale vârfurilor și cu o unitate la suma gradelor exterioare ale vârfurilor.

Fie  $n-1 \geq d_1 \geq d_2 \geq \dots \geq d_n \geq 0$  o secvență descrescătoare de  $n$  numere naturale. Această secvență se numește *secvență grafică* dacă există un graf neorientat cu gradele vârfurilor  $d_1, d_2, \dots, d_n$ .

De exemplu, pentru  $n=7$ , secvența  $4, 2, 2, 2, 2, 1, 1$  este o secvență grafică (pentru că este secvență gradelor vârfurilor grafului din figura următoare).



Dar secvența  $5, 4, 2, 2, 1, 1, 0$  nu este secvență grafică, deoarece nu există nici un graf ale căruia vârfuri să aibă gradele specificate în secvență.

### Teorema 2

Secvența de numere naturale nenule  $D = (d_1 \geq d_2 \geq \dots \geq d_n)$  este secvență grafică dacă și numai dacă secvența  $D' = (d_2 - 1, d_3 - 1, \dots, d_{d_1+1} - 1, d_{d_1+2}, \dots, d_n)$  este secvență grafică și  $d_1 \leq n - 1$ .

### Demonstrație

*Suficiența.* Presupunând că  $D'$  este secvență grafică, deducem că există un graf cu secvența gradelor  $D'$ . Introducem în acest graf un nou vârf pe care îl vom uni prin muchii de vârfurile  $2, 3, \dots, d_1 + 1$ . Obținem astfel un graf cu secvența gradelor  $D$ .

*Necesitatea.* Să considerăm că  $D$  este secvență grafică, prin urmare există cel puțin un graf cu secvența gradelor  $D$ . Să considerăm că  $G$  este un astfel de graf și anume cel pentru care suma gradelor vârfurilor adiacente cu vârful 1 este maximă. Să demonstrăm că în graful  $G$ , vârful 1 este adiacent chiar cu vârfurile  $2, 3, \dots, d_1 + 1$ . (Ca urmare, putem să eliminăm din  $G$  vârful 1 și, evident, toate muchiile incidente cu acesta și să obținem un alt graf  $G'$  cu  $n-1$  vârfuri și secvența gradelor  $D'$ .)

Să presupunem că există un vârf  $k$  ( $2 \leq k \leq d_1 + 1$ ), astfel încât vârful  $k$  nu este adiacent cu vârful 1. Rezultă că există un vârf  $j$  ( $j > d_1 + 1$ ), astfel încât vârfurile 1 și  $j$  sunt adiacente. Dacă gradul  $d_k = d_j$ , putem interschimba vârfurile  $j$  și  $k$ , fără a altera ordinea din secvența gradelor. Dacă  $d_k < d_j$ , cum  $j > k$ , deducem că  $d_j < d_k$ . Dar acest lucru contrazice modul de alegere a grafului  $G$  (înlăturând vârful  $j$  cu vârful  $k$ , am obținut un graf în care suma gradelor vârfurilor adiacente cu 1 este mai mare).

### Observație

Demonstrația acestei propoziții conduce și la un algoritm constructiv de determinare a unui graf cu secvența gradelor dată: la fiecare pas vom uni prin muchii vârful curent  $i$  cu fiecare dintre cele  $d_i$  vârfuri care urmează și vom decrementa gradele acestora; după fiecare pas, trebuie să reordonăm vârfurile după grade. Algoritmul se termină fie când am epuizat vârfurile (după  $n-1$  pași), fie când

găsim un vârf împreună cu care nu îl putem uni cu cele altele următoare (am obținut un grad negativ), caz în care vom concluziona că D nu este secvență grafică.

### Exerciții rezolvate

- Să se determine numărul maxim de muchii într-un graf neorientat cu n vârfuri.

#### Soluție

Numărul maxim de muchii se obține atunci când oricare două vârfuri din graf sunt adiacente. Prin urmare, numărul maxim de muchii într-un graf cu n vârfuri este egal cu numărul de submulțimi de două elemente ale mulțimii  $\{1, 2, \dots, n\}$ , adică  $n(n-1)/2$ .

- Să se determine numărul grafurilor neorientate cu n vârfuri.

#### Soluție

Să notăm cu  $m=n(n-1)/2$  numărul maxim de muchii într-un graf cu n vârfuri și să considerăm că muchiile posibile sunt numerotate de la 1 la m. Într-un graf cu n vârfuri, fiecare dintre cele m muchii poate să aparțină sau nu grafului. Putem astfel asocia, în mod biunivoc, fiecărui graf neorientat cu n vârfuri o funcție  $f: \{1, 2, \dots, m\} \rightarrow \{0, 1\}$  astfel:  $f(i)=1$ , dacă muchia numerotată cu i aparține grafului, respectiv  $f(i)=0$ , în caz contrar. Numărul de grafuri neorientate cu n vârfuri este egal cu numărul de funcții astfel definite, adică  $2^m$ .

### Exerciții propuse

- Să se reprezinte vizual un graf neorientat cu 5 vârfuri, în care fiecare vârf are grad maxim.
- Să se reprezinte vizual un graf neorientat cu 11 vârfuri, cu număr minim de muchii, în care să nu existe vârfuri izolate.
- Să se determine numărul minim de muchii dintr-un graf neorientat cu n vârfuri, care nu conține vârfuri izolate.
- Să considerăm graful orientat din figura 3. Determinați gradul interior și gradul exterior al fiecărui vârf.

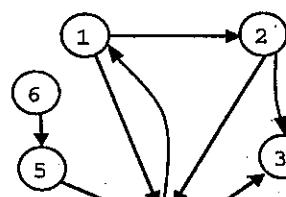


Figura 3

- Să considerăm graful neorientat din figura 4. Determinați gradul fiecărui vârf și identificați vârfurile izolate.

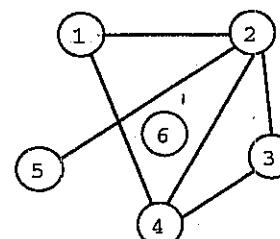


Figura 4

- Demonstrați că în orice graf neorientat, numărul vârfurilor de grad impar este par.
  - Demonstrați că un graf neorientat care are mai mult de  $(n-1)(n-2)/2$  muchii nu conține vârfuri izolate.
  - Demonstrați că într-un graf neorientat, care are cel puțin două vârfuri, există cel puțin două vârfuri cu același grad.
  - Să se determine numărul maxim de arce într-un graf orientat cu n vârfuri.
  - Să se determine numărul de grafuri orientate cu n vârfuri.
  - Un graf neorientat are 80 de noduri și 80 de muchii. Numărul de noduri izolate este cel mult :
- |       |       |       |       |
|-------|-------|-------|-------|
| a. 90 | b. 67 | c. 10 | d. 66 |
|-------|-------|-------|-------|
- (Bacalaureat, iulie 2003)
- Se consideră un graf neorientat cu 8 vârfuri și 15 muchii. Numărul de vârfuri izolate din graf este :
- |            |               |            |               |
|------------|---------------|------------|---------------|
| a. exact 1 | b. cel mult 1 | c. exact 0 | d. cel mult 2 |
|------------|---------------|------------|---------------|
- (Bacalaureat, iulie 2003)
- Să se verifice dacă secvența (7, 4, 2, 1, 1, 1, 1) este sau nu secvență grafică.
  - Scrijeți un program care să verifice dacă o secvență de numere naturale date este sau nu secvență grafică, iar în caz afirmativ să se afișeze muchiile unui graf cu secvența gradelor egală cu secvența specificată.

## Lanț, ciclu, drum, circuit

Se numește *lanț*, într-un graf orientat sau neorientat, o secvență de vârfuri  $[x_1, x_2, \dots, x_p]$ , cu proprietatea că oricare două vârfuri consecutive din secvență sunt adiacente.

De exemplu, pentru graful neorientat din figura 4, secvența  $[1, 4, 2, 5]$  este un lanț. Pentru graful orientat din figura 3, secvența  $[1, 4, 2, 3, 4, 5]$  este un lanț.

Un lanț este numit *elementar* dacă el nu conține de mai multe ori același vârf. Un lanț este *simplu* dacă el nu conține de mai multe ori aceeași muchie.

Lanțul  $[1, 4, 2, 5]$  al grafului din figura 4 este elementar și simplu. Lanțul  $[1, 4, 2, 3, 4, 5]$  al grafului din figura 3 este simplu, dar nu este elementar (trece de două ori prin vârful 4). Lanțul  $[1, 4, 2, 3, 4, 2, 5]$  pentru graful neorientat din figura 4 nu este simplu, deoarece trece de două ori prin muchia  $[2, 4]$ .

Se numește *ciclu*, într-un graf orientat sau neorientat, un lanț simplu pentru care extremitatea inițială coincide cu extremitatea finală.

Se numește *ciclu elementar* un ciclu care nu conține de mai multe ori același vârf (exceptând extremitățile sale).

De exemplu,  $[1, 2, 3, 4, 1]$  este un ciclu elementar în graful neorientat din figura 4.

Se numește *drum* într-un graf orientat o secvență de vârfuri  $(x_1, x_2, \dots, x_p)$ , astfel încât pentru oricare două vârfuri consecutive în secvență  $x_i, x_{i+1}$ , există arcul  $(x_i, x_{i+1})$ .

Drumul se numește *drum elementar* dacă nu conține de mai multe ori același vârf. Drumul se numește *simplu* dacă nu conține de mai multe ori același arc.

De exemplu, pentru graful orientat din figura 3, secvența  $(6, 5, 4, 3)$  este un drum elementar, în timp ce secvența  $(5, 4, 1, 2, 4)$  este un drum în graf care nu e elementar, deoarece conține de două ori vârful 4.

Se numește *circuit* într-un graf orientat un drum simplu, pentru care extremitatea inițială coincide cu cea finală. Circuitul se numește *elementar* dacă nu conține de mai multe ori același vârf (exceptând extremitățile).

De exemplu, pentru graful orientat din figura 3, secvența  $(1, 2, 4, 1)$  este un circuit elementar.

Un lanț/drum/ciclu/circuit elementar se numește *hamiltonian*<sup>1</sup> dacă el trece prin toate vârfurile grafului.

De exemplu, graful din figura 5 conține circuitul hamiltonian:  $(1, 2, 4, 3, 8, 6, 7, 5, 1)$ .

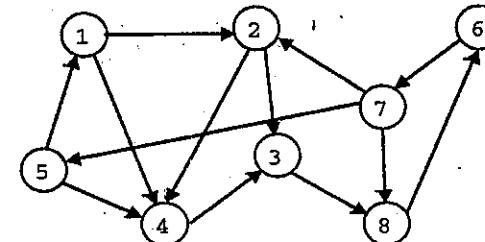


Figura 5

Un lanț/drum/ciclu/circuit se numește *eulerian* dacă trece prin fiecare muchie/arc al grafului exact o dată.

De exemplu, graful din figura următoare conține un ciclu eulerian:  $[1, 2, 3, 7, 6, 8, 7, 4, 5, 2, 4, 1]$ , dar nu conține nici un ciclu hamiltonian.

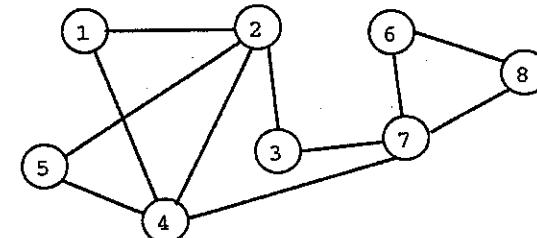


Figura 6

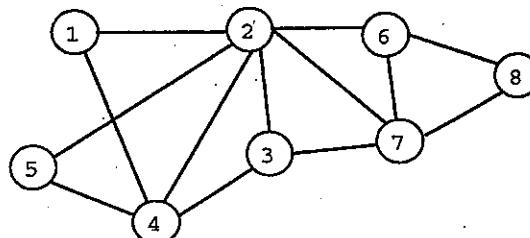
Se numește *lungime* a unui lanț/ciclu/drum/circuit numărul de muchii/arce conținute.

De exemplu, în graful orientat din figura 5, lungimea drumului  $(6, 7, 5, 1)$  este 3, deoarece conține arcele  $(6, 5)$ ,  $(5, 4)$  și  $(4, 3)$ . În graful neorientat din figura 6, lungimea ciclului  $[1, 2, 3, 7, 4, 1]$  este 5, deoarece conține muchiile  $[1, 2]$ ,  $[2, 3]$ ,  $[3, 7]$ ,  $[7, 4]$ ,  $[4, 1]$ .

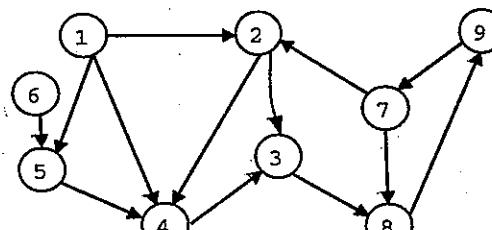
1. Denumirea provine de la numele matematicianului irlandez Sir William Hamilton (1805-1865), care a conceput jocul denumit *Around the World*; scopul jocului era determinarea unui ciclu hamiltonian.

*Exerciții propuse*

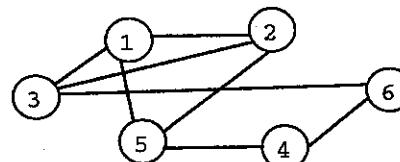
- Construiți un graf cu 5 vârfuri și număr minim de arce, care conține un circuit hamiltonian. Câte astfel de grafuri există?
- Să considerăm graful neorientat din figura următoare. Identificați în acest graf:
  - un lanț elementar de lungime cel puțin egală cu 3;
  - un lanț simplu, care nu este elementar;
  - un lanț care nu este elementar;
  - un ciclu elementar;
  - un ciclu care nu este elementar;
  - un lanț elementar de lungime maximă;
  - un ciclu hamiltonian.



- Să considerăm graful orientat din figura următoare. Identificați în acest graf:
  - un lanț elementar de lungime cel puțin egală cu 4;
  - un drum elementar de lungime cel puțin egală cu 4;
  - un drum care nu este elementar;
  - un ciclu elementar;
  - un circuit elementar;
  - un circuit elementar de lungime maximă;
  - un drum hamiltonian, dacă acesta există.



- Determinați toate ciclurile hamiltoniene ale grafului din figura următoare:



- Dați un exemplu de graf neorientat cu 10 vârfuri și număr maxim de muchii, care să conțină numai lanțuri elementare de lungime cel mult egală cu 2.
- Dați un exemplu de graf neorientat care să conțină cicluri hamiltoniene, dar să nu conțină cicluri euleriene.
- Dați un exemplu de graf orientat care să conțină un drum eulerian, dar să nu conțină nici un drum hamiltonian.
- Să se demonstreze că un graf neorientat care conține lanțuri euleriene are exact două vârfuri de grad impar.
- Se consideră un graf orientat cu 6 vârfuri, etichetate cu numere de la 1 la 6, și 6 arce, astfel încât există un arc de la fiecare vârf cu eticheta  $i$  către un vârf cu eticheta  $i+2$ , dacă există un astfel de nod, sau către nodul cu eticheta  $i-1$ , în caz contrar. Care este lungimea maximă a unui drum în graf?
  - $\infty$
  - 4
  - 3
  - 2

(Simulare Bacalaureat, 2003)

- Dați câte un exemplu de graf orientat cu 10 vârfuri și de un graf orientat cu 11 vârfuri, în care, pentru orice două vârfuri  $x$  și  $y$  din graf, există drum de lungime cel mult egală cu 2 de la  $x$  la  $y$ .
- Scrieți un program care să citească de la tastatură un număr natural nenul  $n$  și să afișeze în fișierul *graf.out* toate arcele unui graf orientat cu  $n$  vârfuri numerotate de la 1 la  $n$ , graf cu proprietatea că între oricare două vârfuri  $x$  și  $y$  ale sale există drum de lungime cel mult egală cu 2 de la  $x$  la  $y$ . Fiecare arc va fi scris pe o linie separată, specificând întâi extremitatea inițială și apoi extremitatea finală.

*Grafuri asociate unui graf dat*

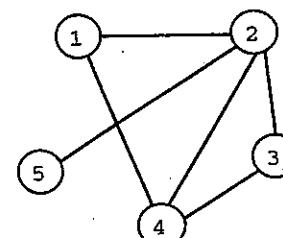
Fie  $G = (V, E)$  un graf orientat sau neorientat. Graful  $G' = (V, E')$  se numește *graf parțial* al grafului  $G$  dacă  $E' \subset E$ .

*Observație*

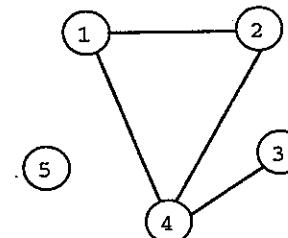
Un graf parțial al unui graf  $G$  se obține eliminând muchii/arce din graful  $G$ .

*Exemplu*

Graful  $G$ :



Un graf parțial al grafului  $G$ , obținut prin eliminarea a două muchii  $\{[2, 5]\} \text{ și } \{[2, 3]\}$ :



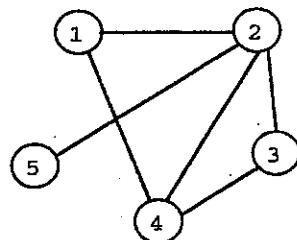
Fie  $G=(V, E)$  un graf orientat sau neorientat. Graful  $G'=(V', E')$  se numește *subgraf al* grafului  $G$  dacă  $V' \subset V$ , iar  $E'$  este mulțimea tuturor muchiilor/arcelor din  $E$  cu proprietatea că au ambele extremități în  $V'$ .

#### *Observație*

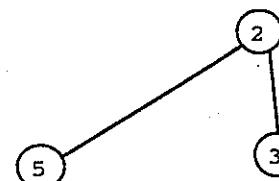
Un subgraf al unui graf  $G$  se obține eliminând vârfuri din  $G$  împreună cu toate muchiile/arcele incidente cu acestea. Se spune că subgraful  $G'$  este induș (sau generat) de mulțimea de vârfuri  $V'$ .

#### *Exemplu*

Graful  $G$ :



Un subgraf al grafului  $G$ , obținut prin eliminarea vârfurilor 1 și 4 și a muchiilor incidente cu acestea:



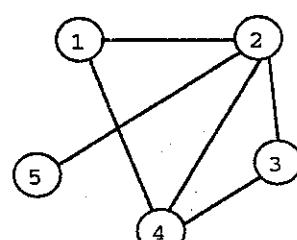
Fie  $G=(V, E)$  un graf orientat sau neorientat. Graful  $G'=(V', E')$  se numește *subgraf parțial* al grafului  $G$ , dacă  $V' \subset V$ , iar mulțimea  $E'$  este inclusă în mulțimea tuturor muchiilor/arcelor din  $E$  cu proprietatea că au ambele extremități în  $V'$ .

#### *Observație*

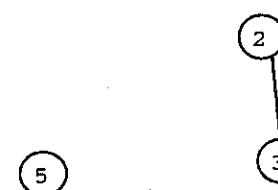
Un subgraf parțial al unui graf  $G$  se obține eliminând vârfuri din  $G$ , toate muchiile/arcele incidente cu vârfurile eliminate, precum și alte muchii/arce din graf.

#### *Exemplu*

Graful  $G$ :



Un subgraf parțial al grafului  $G$ , obținut prin eliminarea vârfurilor 1 și 4, a muchiilor incidente cu acestea și a muchiei {2, 5}:



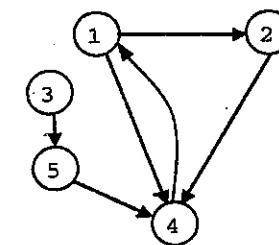
Fie  $G=(V, E)$  un graf orientat. Graful  $G^T=(V, E^T)$  se numește *graful transpus* al grafului  $G$  dacă  $E^T=\{(y, x) | (x, y) \in E\}$ .

#### *Observație*

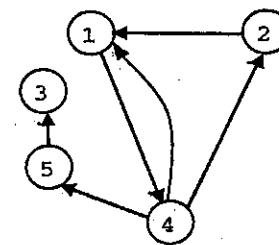
Graful transpus are aceeași mulțime de vârfuri, iar mulțimea arcelor este formată din arcele grafului inițial, cu sensul inversat.

#### *Exemplu*

Graful  $G$ :



Graful transpus  $G^T$ :



#### *Exerciții rezolvate*

1. Fie  $G$  un graf cu  $n$  vârfuri și  $m$  muchii/arce. Determinați numărul de grafuri parțiale ale lui  $G$ .

#### *Soluție*

Să numerotăm muchiile grafului de la 1 la  $m$ . Fiecare graf parțial îl putem asocia în mod biunivoc o funcție  $f : \{1, 2, \dots, m\} \rightarrow \{0, 1\}$  astfel:  $f(i)=1$  dacă muchia numerotată  $i$  aparține grafului parțial, respectiv  $f(i)=0$ , în caz contrar. Numărul de grafuri parțiale este egal cu numărul de funcții astfel definite, adică  $2^m$  (considerăm că un graf este graf parțial al său).

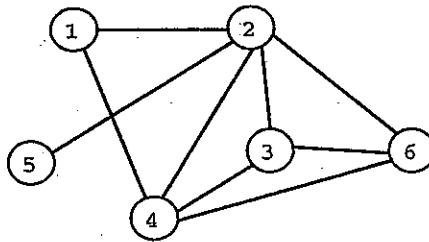
2. Fie  $G$  un graf cu  $n$  vârfuri și  $m$  muchii/arce. Determinați numărul de subgrafuri ale lui  $G$ .

#### *Soluție*

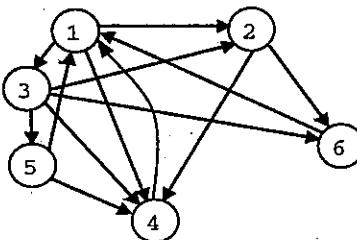
Pentru a genera un subgraf, trebuie să selectăm mulțimea vârfurilor sale, mulțimea muchiilor/arcelor fiind unic determinată de mulțimea vârfurilor selectate. Mulțimea  $\{1, 2, \dots, n\}$  are  $2^n$  submulțimi, dintre care eliminăm mulțimea vidă. Prin urmare, există  $2^n - 1$  subgrafuri ale unui graf cu  $n$  vârfuri (considerăm că un graf este subgraf al său).

*Exerciții propuse*

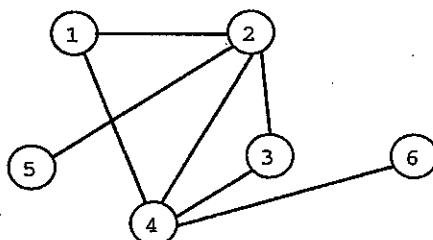
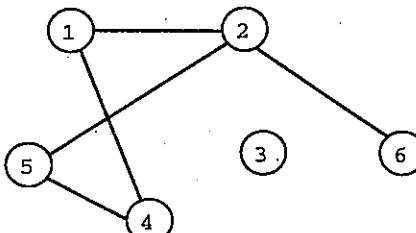
1. Se consideră graful neorientat din figura următoare. Să se determine un subgraf cu număr maxim de vârfuri, cu proprietatea că oricare două vârfuri din subgraf sunt adiacente.



2. Se consideră graful orientat din figura următoare. Să se determine un graf parțial al său cu număr maxim de arce și cu proprietatea că există un vârf care are gradul interior 0 și gradul exterior 3.



3. Se consideră grafurile din figurile următoare. Este graful  $G'$  un graf parțial al grafului  $G$ ? Dacă nu, eliminați un număr minim de muchii din  $G'$ , astfel încât  $G'$  să devină graf parțial al lui  $G$ .

Graful  $G$ :Graful  $G'$ :

4. Să se determine graful transpus al grafului orientat de la exercițiul 2.  
 5. Determinați un graf parțial al grafului neorientat de la exercițiul 1, care să aibă număr maxim de muchii și care să nu conțină cicluri.  
 6. Determinați un subgraf al grafului orientat de la exercițiul 2, care să aibă număr maxim de vârfuri și toate vâfurile să fie izolate.

*Tipuri speciale de grafuri**Graf complet*

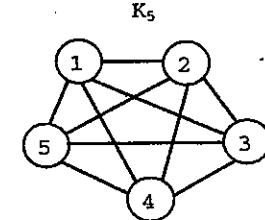
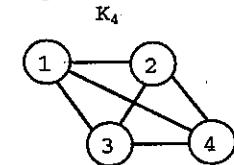
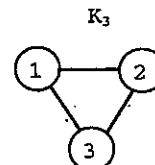
Un graf orientat sau neorientat se numește *complet* dacă oricare două vârfuri din graf sunt adiacente.

*Observație*

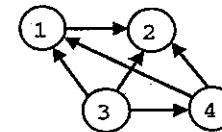
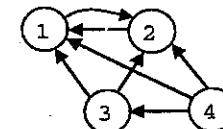
Graful neorientat complet cu  $n$  vârfuri se notează  $K_n$  și conține  $n(n-1)/2$  muchii.

*Exemple*

Grafurile neorientate complete cu  $n=3, 4, 5$  vârfuri:



Pentru un număr de vârfuri fixat, graful neorientat complet este unic, dar grafuri orientate complete există mai multe. De exemplu, iată două grafuri orientate complete cu 4 vârfuri:

Graful  $G_1$ :Graful  $G_2$ :*Graf antisimetric*

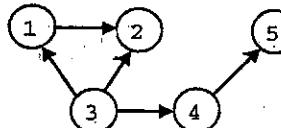
Un graf orientat se numește *antisimetric* dacă pentru orice  $x$  și  $y$ , vârfuri din graf, dacă există arcul  $(x, y)$ , atunci nu există arcul  $(y, x)$ .

*Observație*

Orice relație de ordine între elementele unei mulțimi poate fi modelată cu ajutorul unui graf orientat antisimetric (vâfurile grafului corespund elementelor mulțimii; dacă elementul  $x$  este în relația de ordine respectivă cu elementul  $y$ , atunci în graf va exista arcul  $(x, y)$ ; graful astfel definit este antisimetric, deoarece orice relație de ordine este antisimetrică).

***Exemple***

Graful  $G_1$  din exemplul precedent este antisimetric, dar graful  $G_2$  nu este antisimetric (deoarece există și arcul  $(1, 2)$  și arcul  $(2, 1)$ ). Graful următor este antisimetric, dar nu este complet:

***Graf turneu***

Un graf orientat complet și antisimetric se numește *graf turneu*.

***Exemplu***

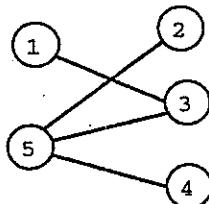
Graful  $G_1$  este complet și antisimetric, deci este graf turneu.

***Graf bipartit***

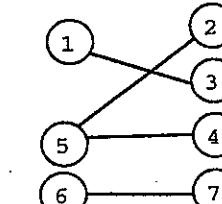
Un graf neorientat  $G=(V, E)$  se numește *bipartit* dacă mulțimea vârfurilor sale poate fi partionată în două submulțimi nevide ( $V=A \cup B$ ,  $A \cap B=\emptyset$ ), astfel încât orice muchie are o extremitate în mulțimea A și cealaltă extremitate în mulțimea B.

***Exemple***

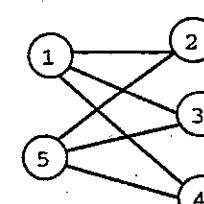
Graful  $G_1$ :



Graful  $G_2$ :



Graful  $G_3$ :

***Graf bipartit complet***

Un graf bipartit se numește *complet* dacă fiecare vârf din mulțimea A este adiacent cu fiecare vârf din mulțimea B.

***Observație***

Dacă numărul de vârfuri din mulțimea A este  $p$ , iar numărul de vârfuri din mulțimea B este  $q$ , graful bipartit complet se notează  $K_{p,q}$  și conține  $p \cdot q$  muchii.

***Exemplu***

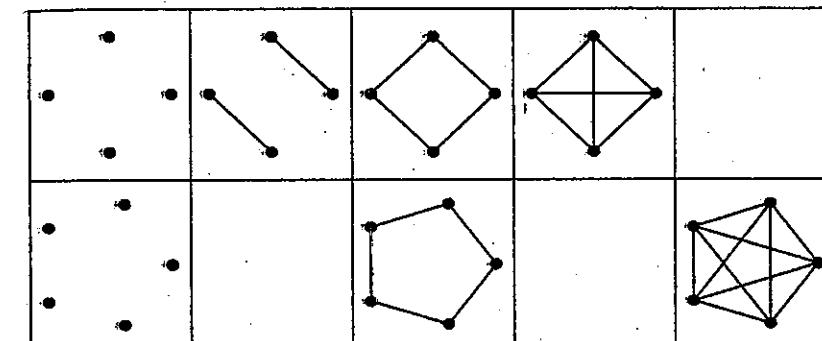
Graful  $G_3$  din exemplul precedent este bipartit complet.

***Graf regulat***

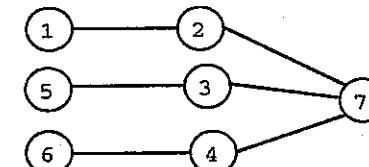
Un graf neorientat se numește *regulat* dacă toate vâfurile sale au același grad.

***Exemple***

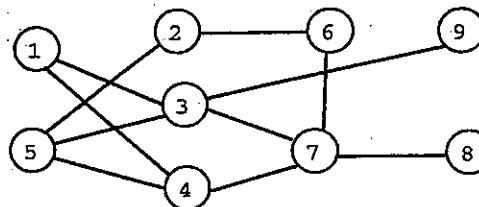
Grafurile regulate cu 4 vârfuri și respectiv 5 vârfuri sunt:

***Exerciții propuse***

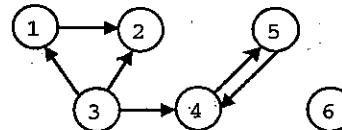
1. Să se descrie grafurile  $K_6$  și  $K_{4,5}$ .
2. Determinați numărul de grafuri orientate complete cu  $n$  vârfuri.
3. Determinați numărul de grafuri turneu cu  $n$  vârfuri.
4. Determinați toate grafurile regulate cu 6 și respectiv 7 vârfuri, făcând abstracție de numerotarea vârfurilor.
5. Care dintre secvențele următoare reprezintă sirul gradelor unui graf neorientat complet cu 5 vârfuri:
  - a. 5, 5, 5, 5, 5
  - b. 5, 4, 3, 2, 1
  - c. 4, 4, 4, 4, 4
  - d. 3, 4, 3, 4, 3
6. Se consideră graful din figura următoare. Care este numărul minim de muchii ce trebuie adăugate astfel încât graful să devină regulat?



7. Determinați numărul de cicluri hamiltoniene din graful  $K_n$ .
8. Determinați numărul de cicluri hamiltoniene din graful bipartit complet  $K_{n,n}$ .
9. Este graful din figura următoare un graf bipartit?



10. Să se construiască un graf bipartit cu 7 vârfuri, care să aibă număr maxim de muchii și să nu conțină cicluri.
11. Este graful din figura următoare un graf turneu? Dacă nu, efectuând un număr minim de modificări (adăugări sau ștergeri de arce), transformați graful în graf turneu.



12. Câte subgrafuri complete cu 3 vârfuri conține un graf complet cu 8 vârfuri?
13. Demonstrați că un graf bipartit cu număr impar de noduri nu este hamiltonian.
14. Demonstrați că un graf care conține cicluri de lungime impară nu este bipartit.
15. Demonstrați că orice graf turneu conține un drum hamiltonian (Rédei, 1934).

## 1.2. Reprezentarea grafurilor în memorie

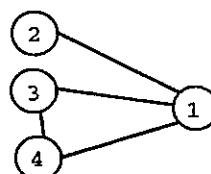
### Reprezentarea prin matrice de adiacență

Fie  $G = (V, E)$  un graf neorientat. Să notăm cu  $n$  numărul de vârfuri din graf.

Matricea de adiacență este o matrice pătratică, având  $n$  linii și  $n$  coloane, cu elemente din mulțimea  $\{0, 1\}$ , astfel:  $A[i][j]=1$ , dacă există muchia  $[i, j]$  în graf și  $A[i][j]=0$ , în caz contrar.

#### Exemplu

Graful  $G_1$ :



Matricea de adiacență a grafului  $G_1$ :

	1	2	3	4
1	0	1	1	1
2	1	0	0	0
3	1	0	0	1
4	1	0	1	0

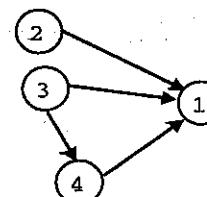
În mod similar, se definește matricea de adiacență a unui graf orientat.

Fie  $G = (V, E)$  un graf orientat. Să notăm cu  $n$  numărul de vârfuri din graf.

Matricea de adiacență este o matrice pătratică, având  $n$  linii și  $n$  coloane, cu elemente din mulțimea  $\{0, 1\}$ , astfel:  $A[i][j]=1$ , dacă există arcul  $(i, j)$  în graf și 0, în caz contrar.

#### Exemplu

Graful  $G_2$ :



Matricea de adiacență a grafului  $G_2$ :

	1	2	3	4
1	0	0	0	0
2	1	0	0	0
3	1	0	0	1
4	1	0	0	0

#### Observații

1. Matricea de adiacență a unui graf neorientat este simetrică față de diagonala principală, în timp ce matricea de adiacență a unui graf orientat nu este simetrică față de diagonala principală.
2. Dimensiunea spațiului de memorie necesar pentru reprezentarea unui graf prin matrice de adiacență este  $O(n^2)$ .

#### Citirea unui graf neorientat și reprezentarea sa prin matrice de adiacență

Din fișierul de intrare `grafn.in` se citesc de pe prima linie două numere naturale  $n$  ( $n \leq 100$ ) și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de muchii dintr-un graf neorientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitățile unei muchii din graf. Să se construiască matricea de adiacență a grafului.

#### Soluție

Considerăm că numărul de vârfuri din graf  $n$  și matricea de adiacență a grafului  $A$  sunt variabile globale (automat inițializate cu 0).

```
#define NMax 101
int n, A[NMax][NMax];
```

Am declarat dimensiunea maximă a matricei cu o unitate mai mare deoarece vom considera că vârfurile sunt numerotate de la 1 la  $n$  și că linia  $i$ , respectiv coloana  $i$  din matrice corespunde vârfului  $i$ .

Vom citi din fișierul de intrare fiecare muchie  $[x, y]$  și vom plasa valoarea 1 în matricea de adiacență, atât pe poziția  $(x, y)$ , cât și pe poziția  $(y, x)$ .

```

void Citire_graf_neorientat()
{
    int m, x, y;
    ifstream fin("grafn.in");
    fin >> n >> m;
    while (m--)
        {fin >> x >> y;
         A[x][y]=A[y][x]=1; }
    fin.close();
}

```

### Citirea unui graf orientat și reprezentarea sa prin matrice de adiacență

Din fișierul de intrare `grafo.in` se citesc de pe prima linie două numere naturale  $n$  și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de arce dintr-un graf orientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitatea initială și extremitatea finală a unui arc din graf. Să se construiască matricea de adiacență a grafului.

#### Soluție

Vom citi din fișierul de intrare fiecare arc  $(x, y)$  și vom plasa valoarea 1 în matricea de adiacență doar pe poziția  $(x, y)$ .

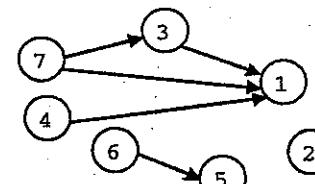
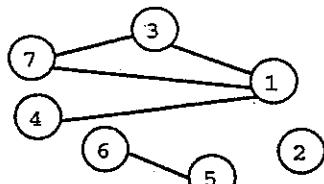
```

void Citire_graf_orientat()
{
    int m, x, y;
    ifstream fin("grafo.in");
    fin >> n >> m;
    while (m--)
        {fin >> x >> y;
         A[x][y]=1; }
    fin.close();
}

```

### Exerciții propuse

1. Construiți reprezentările prin matrice de adiacență ale grafurilor următoare:



2. Matricea următoare este matricea de adiacență a unui graf.

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	0	0	0
3	1	1	0	0	1	1
4	0	0	0	0	0	0
5	0	0	1	0	0	1
6	0	0	1	0	1	0

Care dintre următoarele afirmații sunt adevărate?

- a. Graful este neorientat.
  - b. Graful conține un ciclu eulerian.
  - c. Graful este bipartit.
  - d. Toate vârfurile grafului au grad par.
  - e. Graful este graf turneu.
3. Se consideră graful neorientat dat prin matricea de adiacență următoare. Să se determine lungimea minimă a unui lanț ce unește vârfurile 1 și 3.

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

- a. 2    b. 4    c. 1    d. 3

(Bacalaureat, iulie 2003)

4. Care dintre următoarele este matricea de adiacență a unui graf orientat cu 4 arce?

- |            |            |            |            |
|------------|------------|------------|------------|
| a. 0 0 1 1 | b. 0 1 1 1 | c. 0 1 1 1 | d. 0 1 0 0 |
| 0 0 1 1    | 0 0 0 0    | 0 0 0 0    | 1 0 1 0    |
| 1 1 0 0    | 1 0 0 0    | 1 0 0 1    | 0 0 1 0    |
| 1 1 0 0    | 0 0 0 0    | 0 0 0 0    | 0 0 0 0    |

(Bacalaureat special, 2006)

5. Stabiliti care dintre următoarele matrice de adiacență corespunde grafului din figura următoare:



- |            |              |              |            |
|------------|--------------|--------------|------------|
| a. 0 0 0 1 | b. 0 1 0 1 0 | c. 0 0 0 1 1 | d. 0 1 1 0 |
| 0 0 1 1    | 1 0 0 1 0    | 0 0 0 0 1    | 1 0 1 0    |
| 0 1 0 1    | 0 0 0 0 0    | 0 0 0 0 0    | 1 1 0 1    |
| 1 1 1 0    | 1 1 0 0 0    | 1 0 0 0 1    | 0 0 1 0    |
|            | 0 0 0 1 0    | 1 1 0 1 0    |            |

(Bacalaureat, august 2003)

6. Care este gradul interior al vârfului cu gradul exterior cel mai mare din graful orientat cu matricea de adiacență următoare:

0	0	1	0
0	0	1	1
1	1	0	1
1	1	0	0

- a. 1    b. 2    c. 3    d. 0

(Bacalaureat, august 2003)

7. Care este matricea de adiacență a unui graf neorientat cu 4 vârfuri, două muchii și cel puțin un vârf izolat?

- |          |            |            |            |
|----------|------------|------------|------------|
| a. 0 0 1 | b. 0 1 1 0 | c. 0 1 0 1 | d. 0 0 0 0 |
| 0 0 1    | 1 0 0 0    | 1 0 0 0    | 0 0 1 1    |
| 1 1 0    | 1 0 1 0    | 0 0 0 0    | 0 1 0 1    |
| 0 0 0 0  | 0 0 0 0    | 1 0 0 0    | 0 1 1 0    |

(Simulare Bacalaureat, 2004)

8. Orice graf neorientat cu  $n$  noduri are o matrice de adiacență cu următoarea proprietate:  
 a. Este simetrică față de diagonala principală.  
 b. Este simetrică față de diagonala secundară.  
 c. Este formată numai din valorile 0, 1 și -1.  
 d. Are suma elementelor egală cu  $n$ .
9. Din fișierul de intrare `grafn.in` se citesc de pe prima linie două numere naturale  $n$  și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de muchii dintr-un graf neorientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitățile unei muchii din graf. De pe ultima linie a fișierului de intrare se citește un număr natural  $p$ , urmat de  $p$  numere naturale cuprinse între 1 și  $n$ , reprezentând o secvență de  $p$  vârfuri din graf.  
 a. Să se construiască matricea de adiacență a grafului din fișierul de intrare și să se afișeze în fișierul de ieșire `grafn.out`.  
 b. Să se determine gradul fiecărui vârf din graf și să se afișeze pe o linie separată în fișierul de ieșire `grafn.out`.  
 c. Să se verifice dacă secvența de vârfuri citită de pe ultima linie a fișierului de intrare este un lanț în graf; în caz afirmativ, să se identifice proprietățile lanțului (elementar, simplu) și să se afișeze un mesaj corespunzător rezultatului verificării în fișierul de ieșire.  
 d. Să se verifice dacă graful din fișierul de intrare este graf complet și să se afișeze un mesaj corespunzător rezultatului verificării, pe ultima linie a fișierului de ieșire `grafn.out`.
10. Scrieți un program care să citească două grafuri neorientate date în formatul de la problema precedentă și care să verifice dacă primul graf este un graf parțial al celui de al doilea graf.
11. Din fișierul de intrare `grafo.in` se citesc de pe prima linie două numere naturale  $n$  și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de arce dintr-un graf orientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitatea inițială și extremitatea finală a unui arc din graf. De pe ultima linie a fișierului de intrare se citește un număr natural  $p$ , urmat de  $p$  numere naturale cuprinse între 1 și  $n$ , reprezentând o secvență de  $p$  vârfuri din graf.  
 a. Să se construiască matricea de adiacență a grafului din fișierul de intrare și să se afișeze în fișierul de ieșire `grafo.out`.  
 b. Să se determine gradul interior și gradul exterior al fiecărui vârf din graf și să se afișeze în fișierul de ieșire `grafo.out` (pe linia  $n+1$  gradele interioare separate prin câte un spațiu, iar pe linia  $n+2$  gradele exterioare).  
 c. Să se verifice dacă secvența de vârfuri citită de pe ultima linie a fișierului de intrare este un drum sau un lanț în graf; în caz afirmativ, să se identifice proprietățile drumului/lanțului (elementar, simplu) și să se afișeze un mesaj corespunzător rezultatului verificării în fișierul de ieșire.

- d. Să se verifice dacă graful din fișierul de intrare este graf complet și să se afișeze un mesaj corespunzător rezultatului verificării pe următoarea linie a fișierului de ieșire `grafo.out`.  
 e. Să se verifice dacă graful din fișierul de intrare este graf antisimetric și să se afișeze un mesaj corespunzător rezultatului verificării pe următoarea linie a fișierului de ieșire `grafo.out`.
12. Un vârf al unui graf orientat se numește supersursă dacă el are gradul interior egal cu 0 și gradul exterior egal cu  $n-1$  (unde  $n$  reprezintă numărul de vârfuri din graf). Fiind dat un graf orientat în formatul de la problema precedentă, scrieți un algoritm eficient care să identifice supersursele grafului.

### Reprezentarea prin liste de adiacență

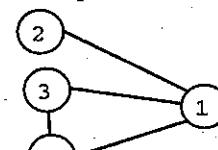
Fie  $G = (V, E)$  un graf neorientat sau orientat cu  $n$  vârfuri.

Pentru a reprezenta graful prin liste de adiacență, vom utiliza un vector cu  $n$  componente, în care vom reține pentru fiecare vârf din graf lista sa de adiacență.

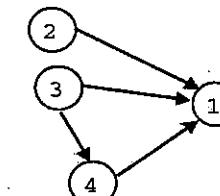
Lista de adiacență a vârfului  $x$  va conține toate vârfurile  $y$  cu proprietatea că există muchia  $[x, y]$  (pentru graf neorientat), respectiv există arcul  $(x, y)$  (pentru graf orientat). Ordinea în care sunt memorate vârfurile într-o listă de adiacență nu contează.

#### Exemplu

Graful  $G_1$ :



Graful  $G_2$ :



Reprezentarea prin liste de adiacență a grafului  $G_1$ :

1	2	3	4	
2	1			
3	1	4		
4	3	1		

Reprezentarea prin liste de adiacență a grafului  $G_2$ :

1			
2	1		
3	1	4	
4	1		

#### Detalii de implementare

##### Varianta I

O listă de adiacență poate fi reprezentată ca un vector în care vârfurile sunt memorate pe poziții consecutive.

Observăm că numărul de elemente din lista de adiacență diferă de la un vârf la altul, fiind egal cu gradul, respectiv gradul exterior al vârfului, grad cuprins între 0 și  $n-1$ .

În primul rând, va fi necesar să memorăm gradul, respectiv gradul exterior al fiecărui vârf. Acest lucru se poate realiza într-un alt vector (vectorul gradelor) sau îl putem memora pe poziția 0 a vectorului în care memorăm lista de adiacență.

Vectorul în care memorăm lista de adiacență a unui vârf poate fi alocat static sau dinamic.

În cazul în care vectorul este alocat static, dimensiunea spațiului de memorie necesar pentru reprezentarea unui graf prin liste de adiacență va fi  $O(n^2)$ , aceeași ca și în cazul matricei de adiacență.

În cazul în care alocăm dinamic memorie pentru vectorul în care este memorată lista de adiacență, dimensiunea spațiului de memorie necesar pentru reprezentarea unui graf prin liste de adiacență va fi  $O(n+m)$ , unde  $m$  este numărul de muchii/arce.

În cazul în care dimensiunea spațiului de memorie alocat static este o problemă, va fi preferată alocarea dinamică a memoriei pentru listele de adiacență.

### Varianta II

Lista de adiacență a unui vârf poate fi implementată cu ajutorul unei structuri de date dinamice, denumită listă înlățuită (această structură o vom studia ulterior).

Dacă structura grafului nu se modifică (deci nu au loc operații de inserare/eliminare a unor muchii/arce din graf), utilizarea unei astfel de implementări nu se justifică, deoarece :

- operațiile de prelucrare a listelor înlățuite sunt mai dificile decât operațiile de prelucrare a vectorilor ;
- se dublează dimensiunea spațiului de memorie necesar, deoarece pentru fiecare nod din lista înlățuită trebuie să reținem și adresa următorului element din listă.

### Citirea unui graf neorientat și reprezentarea sa prin liste de adiacență

Din fișierul de intrare `grafn.in` se citesc de pe prima linie numerele naturale  $n$  ( $n \leq 100$ ) și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de muchii dintr-un graf neorientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitățile unei muchii din graf. Să se construiască reprezentarea prin liste de adiacență a grafului.

### Soluție

Considerăm că numărul de vârfuri din graf  $n$  și reprezentarea prin liste de adiacență a grafului  $A$  sunt variabile globale. De asemenea, vom considera că gradul fiecărui vârf din graf va fi memorat pe poziția 0 a vectorului care reprezintă lista sa de adiacență.

### Implementare statică

```
#define NMax 101
int n, A[NMax][NMax];
```

Vom citi din fișierul de intrare fiecare muchie  $[x, y]$  și vom plasa pe  $y$  în lista de adiacență a lui  $x$ , apoi vom plasa pe  $x$  în lista de adiacență a lui  $y$ .

```
void Citire_graf_neorientat()
{
    int m, x, y;
    ifstream fin("grafn.in");
    fin >> n >> m;
    while (m--)
    {
        fin >> x >> y;
        A[x][0]++;
        A[x][A[x][0]] = y;
        A[y][0]++;
        A[y][A[y][0]] = x;
    }
}
```

### Implementare dinamică

Reprezentarea grafului prin liste de adiacență alocate dinamic va fi un vector cu componente de tip pointer.

```
#define NMax 101
int * A[NMax];
A[i] = adresa de început a vectorului ce va memora lista de adiacență a vârfului i.
```

Înțial, pentru fiecare vârf vom aloca dinamic memorie pentru o singură componentă în vector (componenta 0 în care reținem gradul vârfului).

De fiecare dată când vom citi o muchie  $[x, y]$  :

- vom mări gradul vârfului  $x$ , vom realoca memoria necesară pentru lista de adiacență a lui  $x$  și vom memora vârful  $y$  în lista de adiacență a lui  $x$  ;
- vom mări gradul vârfului  $y$ , vom realoca memoria necesară pentru lista de adiacență a lui  $y$  și vom memora vârful  $x$  în lista de adiacență a lui  $y$ .

Pentru a aloca/realloc dinamic memorie pentru un vector vom utiliza funcțiile `calloc()` și `realloc()`. Aceste funcții sunt declarate în `stdlib.h`:

```
void * calloc(size_t nr, size_t dim);
```

### Efect

Se alocă în mod dinamic un vector cu  $nr$  componente, fiecare componentă având dimensiunea  $dim$  octeți. În cazul alocării cu succes a memoriei, funcția returnează adresa de început a zonei de memorie alocate. În cazul unui eșec (memorie insuficientă,  $nr$  sau  $dim$  nule), funcția returnează valoarea 0 (NULL).

Observați că tipul parametrilor funcției este `size_t`. Acesta este un tip specific dimensiunilor zonelor de memorie.

```
void * realloc(void * adresa_inceput, size_t dim);
```

### Efect

Funcția `realloc()` redimensionează zona de memorie a cărei adresă este specificată în parametrul `adresa_inceput` (această zonă de memorie a fost alocată dinamic în prealabil, prin apelarea funcției `malloc()`, `calloc()` sau `realloc()`). Dacă nu este posibilă stabilirea dimensiunii zonei de memorie la  $dim$  octeți, funcția va căuta o zonă de memorie disponibilă având dimensiunea  $dim$  octeți, va aloca această

zonă de memorie, va copia conținutul fostei zone de memorie în noua locație și va elibera fostă zonă de memorie. În cazul alocării cu succes a memoriei, funcția returnează adresa de început a zonei de memorie alocate (care poate să difere de adresa de început specificată la apel). În cazul unui eșec funcția returnează valoarea 0 (NULL).

Programul următor citește muchiile unui graf neorientat, creează reprezentarea a grafului prin liste de adiacență alocate dinamic, apoi afișează lista de adiacență a fiecărui vârf din graf.

```
#include <stdio.h>
#include <stdlib.h>
#define NMax 101
int n;
int * A[NMax];

void Citire();
void Afisare();

int main()
{
    Citire();
    Afisare();
    return 0;
}

void Citire()
{
FILE * fin=fopen("grafn.in", "r");
int x, y, m, i;
fscanf(fin, "%d %d", &n, &m);
//alloc memorie pentru gradul fiecarui varf
for (i=1; i<=n; i++)
    (A[i]=(int *)realloc(A[i], sizeof(int)));
A[i][0]=0;
for (i=0; i<m; i++)
    fscanf(fin, "%d %d", &x, &y);
    A[x][0]++;
/* incrementam gradul varfului x */
/* reallocam memorie pentru lista de adiacenta a lui x*/
A[x]=(int *)realloc(A[x], (A[x][0]+1)*sizeof(int));
/*memoram pe y in lista de adiacenta a lui x */
A[x][A[x][0]]=y;
A[y][0]++;
/* incrementam gradul varfului y */
/*reallocam memorie pentru lista de adiacenta a lui y*/
A[y]=(int *)realloc(A[y], (A[y][0]+1)*sizeof(int));
/*memoram pe x in lista de adiacenta a lui y */
A[y][A[y][0]]=x;
}

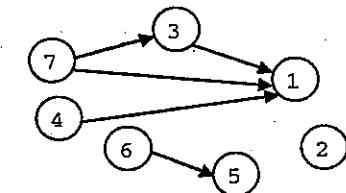
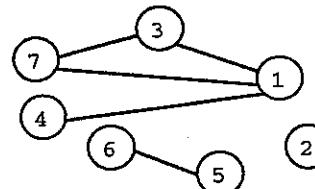
void Afisare()
{
int i, j;
for (i=1; i<=n; i++)
    printf("Lista de adiacenta a varfului %d: ", i);
    for (j=1; j<=A[i][0]; j++)
        printf ("%d ", A[i][j]);
    printf("\n");
}
}
```

### Citirea unui graf orientat și reprezentarea sa prin liste de adiacență

Citirea și reprezentarea unui graf orientat prin liste de adiacență se realizează în mod similar cu reprezentarea grafurilor neorientate prin liste de adiacență. Diferența constă în faptul că la citirea arcului  $(x, y)$  va fi plasat numai vârful  $y$  în lista de adiacență a lui  $x$ , nu și vârful  $x$  în lista de adiacență a lui  $y$ .

### Exerciții propuse

1. Construiți reprezentările prin liste de adiacență ale grafurilor următoare:



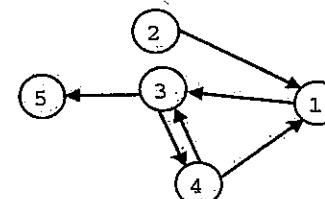
2. Să considerăm următoarea reprezentare prin liste de adiacență a unui graf:

1	2	3	6
2	3		
3	4		
4	2	5	
5			
6	5		

Care dintre următoarele afirmații sunt adevărate?

- a. Reprezentarea este incorectă.
- b. Graful este orientat.
- c. Graful este bipartit.
- d. Graful conține circuite.
- e. Graful este antisimetric.
- f. Graful conține vârfuri izolate.

3. Se consideră graful din figura următoare. Să se determine care dintre liste reprezintă lista de adiacență a vârfului 3.



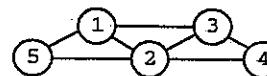
a. 1 4 5

b. 3 1 4 5

c. 4 5

d. 2 4 5

4. Se consideră graful neorientat din figura următoare. Să se determine numărul vârfului care are lista de adiacență  $\{3, 5\}$ .



- a. 5   b. 3   c. 4   d. 1

(Bacalaureat, iulie 2003)

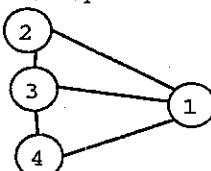
5. Din fișierul de intrare `graf.in` se citesc de pe prima linie două numere naturale  $n$  și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de arce dintr-un graf orientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitățile unui arc.
- Să se reprezinte graful prin liste de adiacență alocate dinamic.
  - Să se determine vârfurile care au gradul interior egal cu gradul exterior.
  - Să se verifice dacă graful este graf turneu.
  - Să se construiască graful transpus al grafului dat și să se afișeze pe ecran lista de adiacență a fiecărui vârf din graful transpus.

### Reprezentarea prin lista muchiilor/arcelor

Pentru a reprezenta un graf neorientat prin lista muchiilor, respectiv un graf orientat prin lista arcelor, se utilizează un vector cu  $m$  componente, unde  $m$  este numărul de muchii/arce din graf. Pentru fiecare muchie/arc vor fi reținute cele două extremități. În cazul muchiilor, ordinea extremităților nu contează. În cazul arcelor, va fi reținută mai întâi extremitatea inițială, apoi extremitatea finală.

#### Exemplu

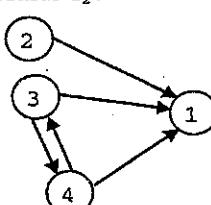
Graful  $G_1$ :



Reprezentarea prin lista muchiilor a grafului  $G_1$ :

1	2	3	4	5
1	1	1	3	2
2	3	4	4	3

Graful  $G_2$ :



Reprezentarea prin lista arcelor a grafului  $G_2$ :

1	2	3	4	5
2	3	3	4	4
1	1	4	3	1

### Detalii de implementare

O muchie/un arc poate fi reprezentată ca o structură cu două câmpuri (câtunul pentru fiecare extremitate), reprezentarea grafului prin lista muchiilor/arcelor fiind astfel un vector de structuri, sau ca un tablou cu două componente, reprezentarea grafului prin lista muchiilor/arcelor fiind astfel o matrice cu două linii și  $m$  coloane.

#### Observații

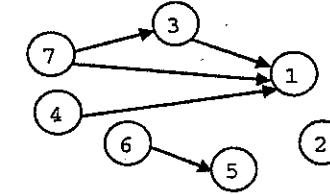
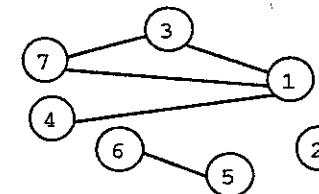
Întrebarea care se impune în acest moment este „care este cea mai bună reprezentare a unui graf?”. Răspunsul este... depinde! Depinde de problema pe care dorim să o rezolvăm. De exemplu, pentru reprezentarea prin matrice de adiacență, timpul de execuție pentru prelucrarea matricei este  $O(n^2)$ , dar răspunsul la o interogare de forma „vârfurile  $x$  și  $y$  sunt adiacente” se obține în  $O(1)$ .

În cazul reprezentării prin liste de adiacență, timpul de execuție pentru prelucrarea matricei este  $O(n+m)$ , dar răspunsul la o interogare de forma „vârfurile  $x$  și  $y$  sunt adiacente” se obține în  $O(n)$ .

Reprezentarea grafului prin lista muchiilor/arcelor poate fi deosebit de utilă pentru problemele în care este necesară parcurgerea muchiilor/arcelor într-o anumită ordine. Dar răspunsul la o interogare de forma „vârfurile  $x$  și  $y$  sunt adiacente” se obține în  $O(m)$ .

### Exerciții propuse

1. Construiți reprezentările prin lista muchiilor/arcelor ale grafurilor următoare:



2. Se consideră următoarea reprezentare prin lista muchiilor a unui graf neorientat cu 7 vârfuri. Să se determine gradele vârfurilor grafului.

1	2	3	4	5	6	7
2	3	3	5	5	7	5
1	2	5	2	7	4	4

3. Se consideră următoarea reprezentare prin lista arcelor a unui graf orientat cu 7 vârfuri. Care dintre următoarele afirmații sunt adevărate?

1	2	3	4	5	6	7	8	9
2	2	3	3	4	7	7	5	5
3	6	4	5	7	5	1	1	6

- Graful conține circuite.
- Vârful 5 are gradul interior egal cu gradul exterior.
- Graful este antisimetric.

- d. Există cel puțin un drum de la vârful 2 la fiecare dintre celelalte vârfuri ale grafului (fiecare vârf din graf este accesibil din vârful 2).  
e. Graful este complet.
4. Din fișierul de intrare `graf.in` se citesc de pe prima linie două numere naturale  $n$  și  $m$ , reprezentând numărul de vârfuri, respectiv numărul de arce dintr-un graf orientat. Se citesc apoi cele  $m$  linii, pe fiecare linie fiind specificate două numere naturale cuprinse între 1 și  $n$ , reprezentând extremitățile unui arc.  
a. Reprezentați graful citit prin lista arcelor.  
b. Determinați gradul interior și gradul exterior al fiecărui vârf.  
c. Verificați dacă graful este antisimetric.  
d. Construiți reprezentarea prin lista arcelor a grafului transpus.

### 1.3. Grafuri ponderate. Reprezentare

În numeroase situații practice modelate cu ajutorul grafurilor, fiecare muchie/arc a/al grafului are asociat un anumit cost sau o anumită pondere (de exemplu, lungimea cablului necesar pentru conectarea a două calculatoare într-o rețea, costul de transport pe o anumită rută, profitul obținut dintr-o anumită tranzacție etc.).

Graful  $G = (V, E)$  (orientat sau neorientat) însotit de o funcție  $c : E \rightarrow \mathbb{R}$ , prin care se asociază fiecărei muchii/arc din graf un număr real se numește *graf ponderat*.

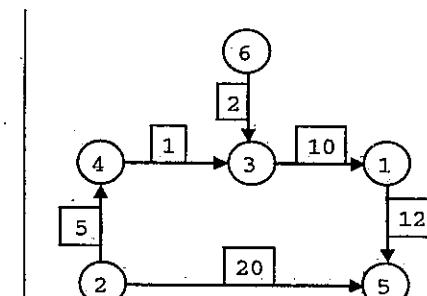
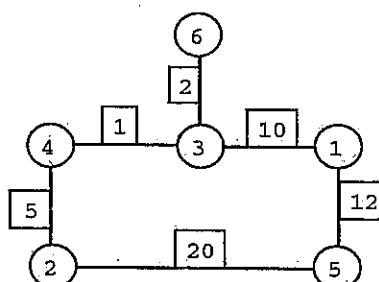
Pentru a reprezenta un graf ponderat trebuie să reținem și costurile asociate muchiilor/arcelor. Astfel, matricea de adiacență devine matricea costurilor, o matrice pătratică  $C$  având  $n$  linii și  $n$  coloane (unde  $n$  este numărul de vârfuri din graf), definită astfel:  $C[x][y]$  va fi costul muchiei/arcului de la  $x$  la  $y$  dacă există, sau 0 dacă  $x=y$ , sau o valoare specială, care depinde de problemă, indicând faptul că nu există muchie/arc de la  $x$  la  $y$ .

În listele de adiacență nu vom reține doar vâfurile adiacente, ci și costurile arcelor/muchiilor corespunzătoare.

În lista muchiilor/arcelor vom adăuga pentru fiecare muchie un câmp suplimentar (costul).

#### Exercițiu propus

Să considerăm grafurile ponderate din figura următoare. Să se reprezinte fiecare graf în cele 3 moduri prezentate.



### 1.4. Închiderea tranzitivă a unui graf

Matricea închiderii tranzitive a unui graf  $G$  este o matrice pătratică  $A^*$  având  $n$  linii și  $n$  coloane (unde  $n$  reprezintă numărul de vârfuri din graf), cu elemente din mulțimea  $\{0, 1\}$ , definită astfel:  $A^*[i][j]=1$  dacă și numai dacă există un drum/lanț de lungime  $>0$  de la  $i$  la  $j$ .

Matricea închiderii tranzitive este cunoscută și sub denumirea de matricea drumurilor (pentru grafuri orientate), respectiv matricea lanțurilor (pentru grafuri neorientate).

Pentru a determina matricea închiderii tranzitive vom utiliza algoritmul *Roy-Warshall*:

1. Se inițializează matricea închiderii tranzitive cu matricea de adiacență a grafului.
2. Considerăm fiecare vârf intermediu  $k$  și verificăm pentru fiecare pereche de vârfuri  $(i, j)$  pentru care  $A^*[i][j]=0$  dacă există drum/lanț de la  $i$  la  $k$  și drum/lanț de la  $k$  la  $j$ ; în caz afirmativ, va exista drum/lanț de la  $i$  la  $j$ , deci  $A^*[i][j]$  devine 1.

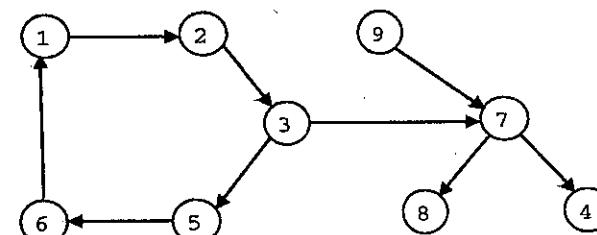
```
void RW()
{
    int k, i, j;
    for (k=1; k<=n; k++)
        for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                AP[i][j]=AP[i][j] || AP[i][k] && AP[k][j];
}
```

#### Observații

1. Complexitatea algoritmului este de  $O(n^3)$ .
2. Pe diagonala principală în matricea închiderii tranzitive se obține valoarea 1 pe linia  $i$  dacă și numai dacă vârful  $i$  aparține unui ciclu/circuit de lungime  $>0$ .
3. Matricea închiderii reflexive și tranzitive a unui graf,  $A^*$ , se definește în mod similar, cu diferența că  $A^*[i][j]=1$  dacă și numai dacă există un drum/lanț de lungime  $\geq 0$  de la  $i$  la  $j$ . Cu alte cuvinte, în matricea închiderii reflexive și tranzitive, diagonala principală trebuie să fie inițializată cu 1.

#### Exerciții propuse

1. Să se determine matricea închiderii tranzitive a grafului din figura următoare:



2. Să considerăm următoarea matrice a închiderii tranzitive a unui graf orientat. Să se identifice vârfurile care nu aparțin nici unui circuit.

```

1 1 1 1 1 1
1 1 1 1 1 1
0 0 0 0 0 1
1 1 1 1 1 1
1 1 1 1 1 1
0 0 0 0 0 0

```

3. Scrieți un program care, utilizând matricea închiderii tranzitive, să determine toate vârfurile accesibile dintr-un vârf dat  $x$ . Spunem că vârful  $y$  este accesibil din vârful  $x$  dacă există un drum de la  $x$  la  $y$  (pentru graf orientat), respectiv un lanț de la  $x$  la  $y$  (pentru graf neorientat).

## 1.5. Parcurgerea grafurilor

Parcurgerea unui graf presupune examinarea sistematică a vârfurilor grafului, cu scopul prelucrării informațiilor asociate vârfurilor.

Există două metode fundamentale de parcugere a grafurilor: parcugerea în adâncime (Depth First Search - DFS) și parcugerea în lățime (Breadth First Search - BFS).

### Parcugerea în adâncime

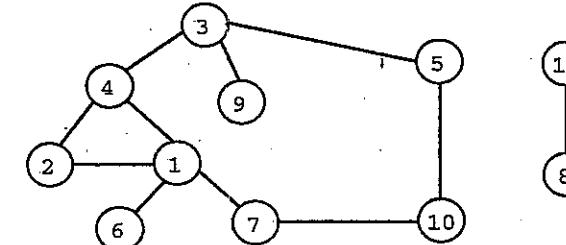
Parcugerea începe cu un vârf inițial, denumit vârf de start. Se vizitează mai întâi vârful de start. La vizitarea unui vârf se efectuează asupra informațiilor asociate vârfului o serie de operații specifice problemei.

Se vizitează apoi primul vecin nevizitat al vârfului de start. Vârful  $y$  este considerat vecin al vârfului  $x$  dacă există muchia  $[x, y]$  (pentru graf neorientat), respectiv arcul  $(x, y)$  (pentru graf orientat).

Se vizitează în continuare primul vecin nevizitat al primului vecin al vârfului de start, și așa mai departe, mergând în adâncime până când ajungem într-un vârf care nu mai are vecini nevizitați. Când ajungem într-un astfel de vârf, revenim la vârful său părinte (vârful din care acest nod a fost vizitat). Dacă acest vârf mai are vecini nevizitați, alegem primul vecin nevizitat al său și continuăm parcugerea în același mod. Dacă nici acest vârf nu mai are vecini nevizitați, revenim în vârful său părinte și continuăm în același mod, până când toate vârfurile accesibile din vârful de start sunt vizitate.

### Exemplu

Să parcugem în adâncime graful din figura următoare, considerând drept vârf de start vârful 3.



Se vizitează mai întâi vârful de start 3. Apoi se vizitează primul vecin nevizitat al lui 3 (ca ordine a vecinilor vom considera ordinea crescătoare a numerelor lor), deci 4. Vizităm apoi primul vecin nevizitat al lui 4, adică pe 1. Apoi vizităm primul vecin nevizitat al lui 1, adică pe 2. În acest moment suntem într-un nod care nu mai are vecini nevizitați, revenim în nodul său părinte, adică în 1. Vârful 1 mai are vecini nevizitați, il vizităm pe primul dintre aceștia, vârful 6. Vârful 6 nu are vecini nevizitați, deci vom reveni în vârful 1, părintele său. Vârful 1 mai are un vecin nevizitat, vârful 7. Vizităm vârful 7, apoi primul vecin nevizitat al lui 7, vârful 10, apoi primul vecin nevizitat al lui 10, vârful 5. Vârful 5 nu mai are vecini nevizitați, deci revenim în 10. Nici vârful 10 nu mai are vecini nevizitați, deci revenim în 7. Nici vârful 7 nu mai are vecini nevizitați, revenim în 1, apoi revenim în 4, apoi în 3. Vârful 3 mai are un vecin nevizitat – vârful 9. Vizităm vârful 9, apoi, deoarece vârful 9 nu are vecini nevizitați, revenim în vârful 3. Cum vârful 3 nu mai are vecini nevizitați și nici părinte (fiind vârful de start), parcugerea s-a încheiat.

Concluzionând, ordinea în care sunt vizitate vârfurile grafului la parcugerea DFS cu vârful de start 3 este: 3, 4, 1, 2, 6, 7, 10, 5, 9.

Vârfurile 8 și 11 nu au fost vizitate, deoarece nu sunt accesibile din vârful 3.

Analizând parcugerea în adâncime, deducem că vârfurile sunt explorate în ordinea inversă a „atingerii” lor, mecanism care poate fi implementat utilizând o stivă. Prin urmare, pentru concizie și claritate se impune o abordare recursivă a parcugerii DFS.

### Reprezentarea informațiilor

1. Graful va fi reprezentat prin liste de adiacență, memorate în tabloul A; pe poziția  $i$  a fiecărei liste de adiacență se află numărul de vârfuri din listă.
  2. Pentru a reține care vârfuri au fost deja vizitate în timpul parcugerii vom utiliza un vector viz, cu  $n$  componente din mulțimea  $\{0, 1\}$ , cu semnificația  $viz[i]=1$  dacă vârful  $i$  a fost deja vizitat, respectiv 0, în caz contrar.
- Considerăm că variabilele  $n$  (numărul de vârfuri din graf), A (listele de adiacență) și viz sunt globale. De asemenea, considerăm că la vizitarea unui vârf va fi afișat pe ecran numărul acestuia.

```

void DFS(int x)
{
    int i;
    //vizitam varful x
    printf("%d ", x);
    viz[x]=1;
    //parcugem lista de adiacenta a varfului x
    for (i=1; i<=A[x][0]; i++)
        if (!viz[A[x][i]])
            //A[x][i] este un vecin nevizitat al lui x
            DFS(A[x][i]);
}

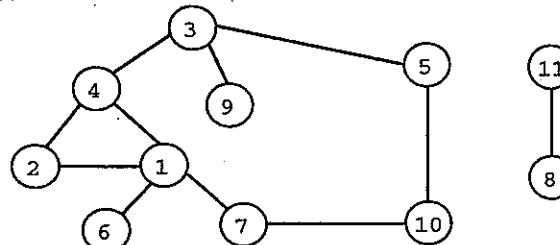
```

### Parcugerea în lătime

Parcugerea în lătime începe, de asemenea, cu un vârf inițial, denumit vârf de start. Se vizitează mai întâi vârful de start. Se vizitează în ordine toți vecinii nevizitați ai vârfului de start. Apoi se vizitează în ordine toți vecinii nevizitați ai vecinilor vârfului de start și așa mai departe, până la epuizarea tuturor vârfurilor accesibile din vârful de start.

#### Exemplu

Să parcugem în lătime graful parcurs deja în adâncime, considerând drept vârf de start vârful 3.



Se vizitează mai întâi vârful de start 3. Apoi se vizitează, în ordine, vecinii nevizitați ai lui 3, deci 4, 5 și 9. Se vizitează apoi, în ordine, vecinii nevizitați ai lui 4 (vârfurile 1 și 2), apoi ai lui 5 (vârful 10) și apoi ai lui 9 (care nu are vecini nevizitați). Se vizitează apoi vecinii vârfului 1 (vârfurile 6 și 7) și parcugerea s-a încheiat (deoarece vârful 2 nu mai are vecini nevizitați, nici vârful 10 și nici vârfurile 6 și 7).

Concluzionând, ordinea în care sunt vizitate vârfurile grafului la parcugerea BFS cu vârful de start 3 este: 3, 4, 5, 9, 1, 2, 10, 6, 7.

Observați că și în cazul parcugerii în lătime vârfurile 8 și 11 nu au fost vizitate, deoarece nu sunt accesibile din vârful 3.

Analizând parcugerea în lătime deducem că vârfurile sunt explorate exact în ordinea „atingerii” lor, mecanism care poate fi implementat utilizând o coadă.

### Descrierea algoritmului

1. Inițializăm coada cu vârful de start și vizităm vârful de start.
2. Cât timp există elemente în coadă executăm :
  - extragem din coadă primul element;
  - parcugem toți vecinii elementului extras, identificându-i pe cei nevizitați; aceștia vor fi vizitați și vor fi plasați în coadă.

### Reprezentarea informațiilor

1. Graful va fi reprezentat prin liste de adiacență, memorate în tabloul A; pe poziția 0 a fiecărei liste de adiacență se află numărul de vârfuri din listă.
2. Pentru a reține care vârfuri au fost deja vizitate în timpul parcugerii, vom utiliza un vector viz, cu n componente din mulțimea {0, 1}, cu semnificația  $\text{viz}[i]=1$  dacă vârful  $i$  a fost deja vizitat, respectiv 0 în caz contrar.
3. Vom utiliza o coadă implementată static într-un vector C cu n elemente, în care reținem vârfurile în ordinea vizitării lor. Variabile prim și ultim rețin poziția de început, respectiv poziția de sfârșit în coadă.

Considerăm că variabilele n (numărul de vârfuri din graf), A (listele de adiacență), C (coada) și viz sunt globale. De asemenea, considerăm că la vizitarea unui vârf va fi afișat pe ecran numărul acestuia.

```

void BFS(int x)
{
    int i, prim, ultim;
    //vizitam varful de start
    printf("%d ", x);
    viz[x]=1;
    //initializam coada cu varful de start
    C[0]=x; prim=ultim=0;
    while (prim<=ultim) //cat timp coada nu este vida
        //extragem un element din coada
        x=C[prim++];
        //parcugem lista de adiacenta a varfului x
        for (i=1; i<=A[x][0]; i++)
            if (!viz[A[x][i]])
                //A[x][i] este un vecin nevizitat al lui x
                //il vizitam
                printf("%d ", A[x][i]);
                viz[A[x][i]]=1;
                C[++ultim]=A[x][i]; //il plasam in coada
}
}

```

### Observații

1. Parcugerea în lătime are o proprietate remarcabilă: fiecare vârf este vizitat pe cel mai scurt drum/lanț începând din vârful de start.
2. Complexitatea timp a algoritmilor de parcugere în adâncime și în lătime a unui graf depinde de modalitatea de reprezentare a acestuia. În cazul reprezentării prin liste de adiacență, complexitatea este  $O(n+m)$ . În cazul reprezentării prin matrice de adiacență, complexitatea este  $O(n^2)$ .

## Aplicații

### Determinarea celui mai scurt drum/lanț între două vârfuri

Fie  $G$  un graf (orientat sau neorientat) și  $x, y$  două vârfuri din graf. Să se determine cel mai scurt drum (respectiv lanț, pentru cazul în care graful este neorientat) de la  $x$  la  $y$ .

#### Soluție

Vom utiliza proprietatea parcurgerii în lățime de a vizita fiecare vârf pe cel mai scurt drum/lanț care pleacă din vârful de start. Prin urmare, pentru a determina cel mai scurt drum de la  $x$  la  $y$ , vom efectua o parcurgere în lățime începând din  $x$ , până când atingem vârful  $y$ , sau până când vizităm toate vâfurile accesibile din  $x$ .

Pentru a reconstituia drumul, vom modifica semnificația vectorului  $viz$ . Mai exact,  $viz[i]=j$  dacă  $j$  este vârful său (memorat în  $viz[y]$ ) și  $i$  este vârful său (memorat în  $viz[i]$ ). Dacă  $viz[i]=-1$ , atunci vârful  $i$  nu a fost vizitat prin parcurgerea arcului  $(j, i)$  (sau a muchiei  $[j, i]$ ), respectiv 0 dacă vârful  $i$  nu a fost vizitat. Singurul vârf vizitat care nu are vârful său este vârful de start. Prin convenție, stabilim  $viz[x]=-1$ .

Funcția de parcurgere în lățime, modificată astfel încât să permită reconstituirea celui mai scurt drum/lanț de la  $x$  la  $y$  este:

```
void BFS(int x)
{
    int i, prim, ultim;
    viz[x]=-1; //vizitam varful de start
    C[0]=x; prim=ultim=0; //initializam coada
    while (prim<=ultim && !viz[y])
        //cat timp coada nu este vida si nu am vizitat varful y
        (x=C[prim++]); //extragem un element din coada
        //parcurgem lista de adiacenta a varfului x
        for (i=1; i<=A[x][0]; i++)
            if (!viz[A[x][i]])
                //A[x][i] este un vecin nevizitat al lui x
                (viz[A[x][i]]=x); //il vizitam
                C[++ultim]=A[x][i]; //il plasam in coada
}
}
```

Reconstituirea drumului/lanțului se face în sens invers, pornind de la vârful  $y$ , determinând apoi său (memorat în  $viz[y]$ ), apoi său (memorat în  $viz[viz[y]]$ ) și aşa mai departe, până când întâlnim un vârf care nu are său (acesta este vârful de start  $x$ ).

Pentru a afișa drumul în ordinea firească, îl vom memora într-un vector auxiliar, denumit  $drum$ , apoi vom afișa vectorul de la sfârșit către început.

```
void Afisare_Drum()
{
    int poz=0, i, drum[NMax];
    if (!viz[y])
        printf("Varful %d nu este accesibil din %d\n", y, x);
    else
        (drum[0]=y);
        while (viz[drum[poz]]!= -1)
            drum[++poz]=viz[drum[poz-1]];
        for (i=poz; i>=0; i--)
            printf("%d ", drum[i]);
        printf("\n");
}
```

#### Bipartit

Fie  $G$  un graf neorientat. Să se verifice dacă graful  $G$  este bipartit.

#### Soluție

Vom parcurge toate vâfurile grafului, încercând să le separăm în două mulțimi, astfel încât orice muchie din graf să aibă o extremitate în prima mulțime și cealaltă extremitate în cea de-a doua mulțime.

Mai exact, vom alege un vârf din graf ca vârful de start și îl vom plasa în prima mulțime. Toți vecinii vârfului de start trebuie să fie plasați în cea de-a două mulțime. Vecinii vecinilor vârfului de start vor fi plasați în prima mulțime și aşa mai departe.

Dacă în timpul parcurgerii un vârf care a fost deja vizitat și astfel a fost plasat în una dintre mulțimi este vizitat din nou, iar la noua vizitare vârful ar trebui plasat în cealaltă mulțime, deducem că graful nu este bipartit.

Este posibil ca nu toate vâfurile grafului să fie accesibile din vârful de start ales. Prin urmare, dacă mai există vâfururi nevizitate, reluăm parcurgerea până când toate vâfurile sunt vizitate (sau am detectat faptul că graful nu este bipartit), alegând de fiecare dată ca vârful de start unul dintre vâfurile nevizitate.

Pentru a reține modul în care vâfurile grafului sunt partionate în cele două mulțimi, vom modifica semnificația vectorului  $viz$  astfel:  $viz[i]=1$ , dacă vârful  $i$  a fost vizitat și plasat în prima mulțime,  $viz[i]=2$ , dacă vârful  $i$  a fost vizitat și plasat în cea de-a două mulțime și respectiv  $viz[i]=0$ , dacă  $i$  nu a fost vizitat.

Pentru a parcurge vâfurile grafului putem utiliza oricare dintre cele două metode de parcurgere.

Vom modifica funcția de parcurgere astfel încât să returneze valoarea 1 dacă multimea vâfurilor accesibile din vârful de start generează un subgraf bipartit și respectiv 0 în caz contrar.

```
int BFS(int x)
{
    int i, prim, ultim;
    //vizitam varful de start
    viz[x]=1;
    //initializam coada cu varful de start
    C[0]=x; prim=ultim=0;
    while (prim<=ultim)
        //cat timp coada nu este vida
        //extragem un element din coada
        x=C[prim++];
```

```

//parcurgem lista de adiacenta a varfului x
for (i=1; i<=A[x][0]; i++)
    if (!viz[A[x][i]])
        //A[x][i] este un vecin nevizitat al lui x
        //il plasam in cealalta multime decat x
        viz[A[x][i]]=3-viz[x];
        //il plasam in coada
        C[++ultim]=A[x][i];
    }
else
    /* varful este vizitat, verificam daca este
       plasat in aceeasi multime cu x */
    if (viz[A[x][i]]==viz[x])
        return 0;
}
return 1;
}

```

În funcția main() apelăm funcția de parcurgere pentru fiecare vârf nevizitat:

```

int x, ok=1;
for (x=1; x<=n && ok; x++)
    if (!viz[x])
        ok=ok && BFS(x);
if (ok) printf("Graful este bipartit\n");
else printf("Graful nu este bipartit\n");

```

### Aciclic

Fie  $G$  un graf neorientat. Să se verifice dacă graful  $G$  este aciclic (nu conține cicluri).

### Soluție

O primă soluție ar fi să construim matricea închiderii tranzitive a grafului  $G$  și apoi să verificăm dacă am obținut valoarea 1 pe diagonala principală (caz în care graful conține cicluri). O astfel de abordare are complexitatea  $O(n^3)$ .

O soluție mai eficientă este de a reprezenta graful prin liste de adiacență și de a realiza o parcurgere în adâncime a grafului.

Dacă în timpul parcurgerii în adâncime întâlnim un vecin  $y$  al vârfului curent  $x$  care a mai fost deja vizitat, există două cazuri posibile :

- $y$  este vârful părinte al vârfului curent  $x$ ;
- $y$  este un vârf care a fost vizitat în prealabil (deci există un lanț de la  $y$  la  $x$ ) și există muchie de la  $x$  la  $y$ ; în concluzie, din lanțul  $[y, \dots, x]$  și muchia  $[x, y]$  se formează un ciclu.

Vom modifica funcția de parcurgere în adâncime pentru a detecta ciclurile astfel :

- funcția va returna valoarea 1 dacă mulțimea vârfurilor accesibile din vârful de start generează un subgraf aciclic și respectiv 0 în caz contrar;
- funcția va avea un parametru suplimentar (pe lângă vârful curent, vom transmite ca parametru și vârful părinte al vârfului curent, pentru a putea face diferență între cele două cazuri posibile).

```

int DFS(int x, int px)
{
    int i;
    viz[x]=1;
    for (i=1; i<=A[x][0]; i++)
        if (!viz[A[x][i]])
            if (!DFS(A[x][i], x)) return 0;
        else
            if (px!=A[x][i]) return 0;
    return 1;
}

```

Deoarece este posibil ca nu toate vârfurile grafului să fie accesibile din vârful de start, în funcția main() va trebui să apelăm funcția de parcurgere de mai multe ori, o dată pentru fiecare vârf nevizitat identificat :

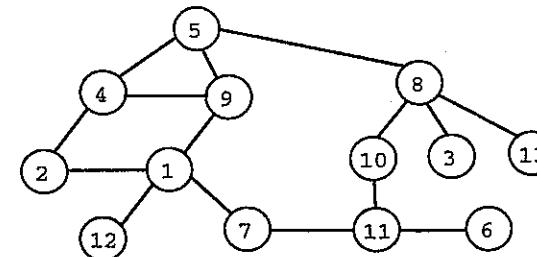
```

int x, ok=1;
for (x=1; x<=n && ok; x++)
    if (!viz[x])
        ok=ok && DFS(x, -1);
if (ok) printf("Graful este aciclic\n");
else printf("Graful nu este aciclic\n");

```

### Exerciții propuse

1. Se dă graful din figura următoare. Să se parcurgă acest graf în adâncime și în lățime începând din vârful 5.



2. Dați exemplu de un graf pentru care parcurgerea în lățime coincide cu parcurgerea în adâncime (ambele parcurgeri având drept vârf de start vârful 1).
3. Să considerăm următoarele secvențe de vârfuri obținute prin parcurgerea în adâncime, respectiv în lățime a unui graf orientat cu 7 vârfuri. Dați exemplu de un graf orientat prin a căruia parcurgere în adâncime și apoi în lățime să obținem secvențele specificate.

Parcurgere în adâncime : 1, 2, 3, 5, 7, 6, 4.

Parcurgere în lățime : 1, 2, 3, 4, 5, 6, 7.

4. Să considerăm că, după o parcurgere în lățime modificată astfel încât vectorul viz să permită reconstituirea celui mai scurt drum de la vârful 7 la vârful 2, conținutul vectorului viz este:

2	-1	5	1	2	3	8	3	8
1	2	3	4	5	6	7	8	9

Care este lungimea celui mai scurt drum de la 7 la 2?

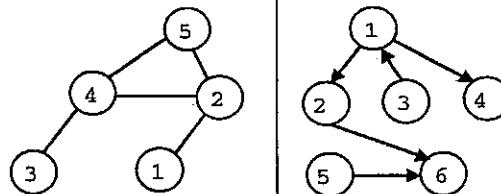
5. Se consideră un graf neorientat complet reprezentat prin matrice de adiacență. Care dintre următoarele afirmații sunt adevărate:
- Parcurgerea în lățime coincide cu parcurgerea în adâncime a grafului, indiferent de vârful de start.
  - Pentru a parcurge toate vârfurile grafului, funcția de parcurgere în lățime trebuie apelată de mai multe ori.
  - Parcugând graful în adâncime, începând din vârful 1, se obțin vârfurile în ordinea crescătoare a numerelor lor.
6. Implementați parcurgerea în adâncime și parcurgerea în lățime a unui graf reprezentat prin matrice de adiacență.
7. Testați dacă un graf este bipartit utilizând parcugerea în adâncime.
8. Scrieți o funcție care să determine numărul de vârfuri accesibile dintr-un vârf specificat al unui graf neorientat.

## 1.6. Conexitate

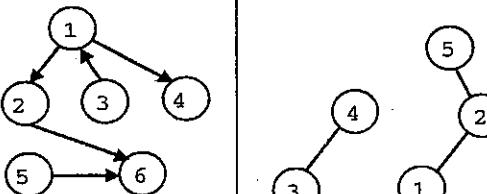
Un graf se numește *conex* dacă oricare ar fi  $x$  și  $y$  vârfuri din graf există lanț între  $x$  și  $y$ .

### Exemple

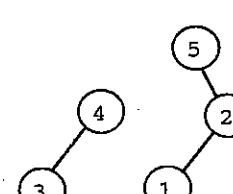
Graf neorientat conex:



Graf orientat conex:



Graf neorientat neconex (de exemplu, între vârfurile 1 și 3 nu există lanț):



Se numește *componentă conexă* un subgraf conex maximal cu această proprietate (adică, dacă am mai adăuga un vârf și toate muchiile/arcele incidente cu acesta, subgrupul obținut nu ar mai fi conex).

### Observații

- Orice graf neconex conține cel puțin două componente conexe. De exemplu, graful neorientat neconex din exemplul precedent are exact două componente conexe: subgrafurile induse de multimile  $\{3, 4\}$  și  $\{1, 2, 5\}$ .
  - Componentele conexe ale unui graf sunt disjuncte.
- Demonstrație.* Să presupunem prin reducere la absurd că există  $C_1$  și  $C_2$ , două componente conexe, astfel încât  $C_1 \cap C_2 \neq \emptyset$ . Fie  $x$  un vârf care aparține intersecției celor două componente conexe. Considerăm  $C_1$  componenta conexă cea mai amplă și  $y$  un vârf din  $C_1 - C_2$ . Deoarece  $x$  și  $y$  aparțin componentei  $C_1$ , deducem că există lanț de la  $y$  la  $x$ . Deoarece  $x$  aparține și componentei  $C_2$ , deducem că există lanț de la  $x$  la orice alt vârf din  $C_2$ . Prin urmare, există lanț de la  $y$  la orice alt vârf din  $C_2$ . Cu alte cuvinte,  $C_2$  nu este maximal (am putea adăuga vârful  $y$ , subgraful rămânând conex), ceea ce este în contradicție cu presupunerea făcută.
- Componentele conexe ale grafului constituie o partiție a mulțimii vârfurilor grafului.

### Descompunerea unui graf neorientat în componente conexe

A descompune un graf în componente conexe înseamnă a determina toate componentele conexe ale grafului.

A determina componenta conexă a unui vârf  $x$  presupune a determina toate vârfurile accesibile din vârful  $x$ ; deci este suficient să realizăm o parcugere a grafului (în lățime sau în adâncime) cu vârful de start  $x$ .

Pentru a descompune graful în componente conexe, vom realiza câte o parcugere pentru fiecare componentă conexă (selectând ca vârf de start vârful nevizitat având număr minim).

### Observații

- Pentru a testa dacă un graf este aciclic sau bipartit, am testat fiecare componentă conexă a grafului.
- Pentru a descompune un graf orientat în componente conexe, se va face abstracție de orientarea arcelor.
- Pentru un graf reprezentat prin liste de adiacență, descompunerea în componente conexe utilizând parcugerea grafului are complexitatea  $O(n+m)$ . Dacă graful este reprezentat prin matrice de adiacență, descompunerea în componente conexe utilizând parcugerea grafului are complexitatea  $O(n^2)$ .

### Descompunerea în componente conexe a unui graf neorientat reprezentat prin lista muchiilor

Dacă graful este reprezentat prin lista muchiilor, descompunerea grafului în componente conexe utilizând parcugerea grafului este ineficientă.

Vom considera că, inițial, graful nu conține nici o muchie, deci este format din  $n$  vârfuri izolate, fiecare vârf formând o componentă conexă.

Vom parcurge lista muchiilor grafului, adăugând în graf muchiile una câte una.

La adăugarea unei muchii pot apărea două cazuri:

- extremitățile muchiei sunt în aceeași componentă conexă (în acest caz, prin adăugarea acestei muchii nu se modifică descompunerea în componente conexe a grafului);
- extremitățile muchiei sunt în componente conexe diferite (în acest caz, prin adăugarea acestei muchii, componentele conexe corespunzătoare celor două extremități se unifică).

Vom reține evidența componentelor conexe ale grafului cu ajutorul unui vector  $C$  cu  $n$  componente,  $C[i]$  = reprezentantul componentei conexe din care face parte vârful  $i$  (prin convenție, vom considera drept reprezentant vârful cu numărul cel mai mic). Inițial:  $C[i] = i$ , pentru orice  $i = 1, 2, \dots, n$ .

```
#include <stdio.h>
#define NMax 101
#define MMax NMax*(NMax-1)/2

typedef struct {int x, y;} Muchie;
int n, m;
Muchie G[MMax];
int C[NMax];

void Citire();
void Descompunere_Comp_Conexe();
void Afisare();

int main()
{
    Citire();
    Descompunere_Comp_Conexe();
    Afisare();
    return 0;
}

void Citire()
{
    FILE * fin=fopen("graf.in", "r");
    int i;
    fscanf(fin, "%d %d", &n, &m);
    for (i=0; i<m; i++)
        fscanf(fin, "%d %d", &G[i].x, &G[i].y);
}

void Descompunere_Comp_Conexe()
{
    int i, j, min, max;
    for (i=1; i<=n; i++) C[i]=i;
    for (j=0; j<m; j++)
        if (G[j].x!=G[j].y)
```

```
/* extremitatile muchiei j nu sunt in aceeasi
   componenta conexa */
{
    if (C[G[j].x]<C[G[j].y])
        min=C[G[j].x], max=C[G[j].y];
    else
        min=C[G[j].y], max=C[G[j].x];
    //unific componentele conexe ale extremitatilor
    for (i=1; i<=n; i++)
        if (C[i]==max) C[i]=min;
}
}

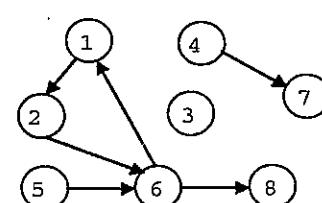
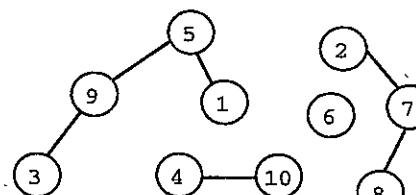
void Afisare()
{
    int nrc=0, i, j;
    for (i=1; i<=n; i++)
        if (C[i])
            {nrc++;
             printf("Componenta conexa %d: %d", nrc, i);
             for (j=i+1; j<=n; j++)
                 if (C[j]==C[i])
                     {printf(" %d", j);
                      C[j]=0;}
             printf("\n");}
}
```

#### Observație

Complexitatea algoritmului de descompunere în componente conexe a unui graf neorientat reprezentat prin lista muchiilor este  $O(n \cdot m)$ . Vom oferi în secțiunile următoare o implementare mai eficientă.

#### Exerciții propuse

- Să se descompună grafurile următoare în componente conexe:



- Să considerăm graful reprezentat prin următoarea matrice de adiacență. Câte componente conexe are acest graf?

0	1	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1	0	1
0	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	0

3. Să considerăm graful reprezentat prin următoarele liste de adiacență. Câte componente conexe are acest graf?

1	2	7	
2	3		
3			
4	6		
5			
6			
7	8	2	3
8			

4. Să considerăm următorul graf neorientat cu  $n=17$  vârfuri reprezentat prin lista muchiilor. Care sunt componentele conexe ale acestui graf?

1	3	1	3	1	5	14	16	15	2	2	9	9	2	7
4	8	3	4	8	10	15	17	16	13	7	11	12	9	12

5. După executarea funcției de descompunere în componente conexe a unui graf neorientat reprezentat prin lista muchiilor, vectorul C în care este memorată evidența componentelor conexe are următorul conținut :

1	2	3	4	5	1	2	1	5	5	2	4	2	5
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Determinați componentele conexe ale grafului.

6. Care este numărul minim de muchii pe care trebuie să le conțină un graf neorientat cu 21 de vârfuri astfel încât, oricum ar fi dispuse aceste muchii, graful să fie conex?

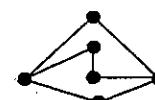
a. 210      b. 11      c. 191      d. 171

7. Într-un graf neorientat cu 10 noduri, fiecare vârf are gradul 2. Care este numărul maxim de componente conexe din care poate fi format graf?

a. 1      b. 3      c. 4      d. 5

(Bacalaureat, iulie 2006)

8. Numărul minim de muchii ce se pot alege pentru a fi eliminate din graful neorientat următor astfel încât acesta să devină neconex este :



a. 4

b. 3

c. 2

d. 1

(Bacalaureat, iulie 2003)

9. Gradul maxim al unui vârf ce se poate obține într-un graf neorientat conex cu  $n$  vârfuri și  $n-1$  muchii este :

a. 2

b.  $n/2$

c.  $n$

d.  $n-1$

(Bacalaureat, iulie 2003)

10. Se consideră un graf cu 10 vârfuri și 6 componente conexe. Care este numărul maxim de muchii din graf?

11. Numărul minim de noduri dintr-un graf neorientat cu 12 muchii, fără noduri izolate, graf format din exact 3 componente conexe este :

a. 7

b. 8

c. 9

d. 10

(Bacalaureat, iulie 2003)

12. Scrieți program care să adauge un număr minim de muchii într-un graf neorientat neconex, astfel încât graful obținut să fie conex. Programul va afișa pe ecran muchiile adăugate.

13. Scrieți un program care să realizeze descompunerea în componente conexe a unui graf neorientat pe baza matricei închiderii tranzitive a grafului.

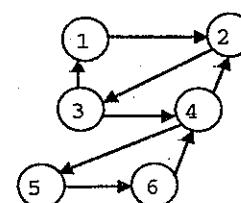
14. Modificați algoritmul de descompunere în componente conexe a unui graf neorientat reprezentat prin lista muchiilor, astfel încât să verifice dacă graful este aciclic.

## 1.7. Tare-conexitate

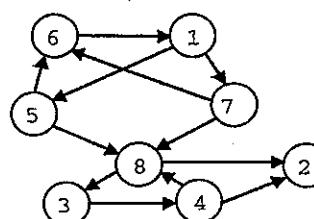
Un graf orientat se numește *tare-conex* dacă oricare ar fi  $x$  și  $y$  vârfuri din graf există drum de la  $x$  la  $y$  și drum de la  $y$  la  $x$ .

### Exemple

Graf orientat tare-conex :



Graf orientat care nu este tare-conex (de exemplu, de la vârful 1 la 8 există drum, dar de la 8 la 1 nu există) :



Se numește *componentă tare-conexă* un subgraf tare-conex maximal cu această proprietate (adică dacă am mai adăuga un vârf și toate arcele incidente cu acesta, subgraful obținut nu ar mai fi tare-conex).

*Observație*

Componentele tare-conexe constituie o parte a mulțimii vârfurilor grafului.

*Descompunerea unui graf orientat în componente tare-conexe*

A descompune un graf în componente tare-conexe înseamnă a determina toate componentele tare-conexe ale grafului.

Componentele tare-conexe ale celui de-al doilea graf din exemplul precedent sunt subgrafurile generate de mulțimile de vârfuri: {1, 5, 6, 7}, {3, 4, 8} și {2}.

*Algoritmul Kosaraju-Sharir (1978)*

1. Se parcurge graful în adâncime și se numerotează vârfurile grafului în postordine (vârful  $x$  este numerotat după ce toți succesorii săi au fost numerotați); în vectorul postordine se memorează ordinea vârfurilor.
2. Se determină graful transpus  $G^T$ .
3. Se parcurge graful transpus în adâncime, considerând vârfurile în ordinea inversă a vizitării lor în parcurgerea DFS a grafului inițial.
4. Fiecare subgraf obținut în parcurgerea DFS a grafului transpus reprezintă o componentă tare-conexă a grafului inițial.

```
#include <stdio.h>
#include <stdlib.h>
#define NMax 101
int n, nr, nrc;
int * A[NMax], * AT[NMax];
int postordine[NMax], viz[NMax];

void Citire();
void DFS(int);
void DFST(int);

int main()
{int i;
 Citire();
 //parcugem graful DFS, determinand ordinea varfurilor
 for (i=1; i<=n; i++)
   if (!viz[i]) DFS(i);
 /* parcugem DFS graful transpus, prelucrand varfurile in
 postordine */
 for (i=n; i>0; i--)
   if (viz[postordine[i]])
     /* componenta tare-conexa din care face parte varful
       curent nu a fost determinata */
     (printf("Componenta tare-conexa %d: ", ++nrc),
      DFST(postordine[i]),
      printf("\n"));
}
return 0;
}
```

```
void Citire()
{//construim graful si graful transpus
FILE * fin=fopen("graf.in","r");
int x, y, m, i;
fscanf(fin,"%d %d", &n, &m);
//aloc memorie pentru gradul fiecarui varf
for (i=1; i<=n; i++)
  {A[i]=(int *)realloc(A[i],sizeof(int));
   A[i][0]=0;
   AT[i]=(int *)realloc(AT[i],sizeof(int));
   AT[i][0]=0;
  }
for (i=0; i<m; i++)
  {fscanf(fin,"%d %d", &x, &y);
   A[x][0]++;
   A[x]=(int *)realloc(A[x], (A[x][0]+1)*sizeof(int));
   A[x][A[x][0]]=y;
   AT[y][0]++;
   AT[y]=(int *)realloc(AT[y], (AT[y][0]+1)*sizeof(int));
   AT[y][AT[y][0]]=x;
  }
}

void DFST(int x)
{int i;
 viz[x]=0; printf(" %d", x);
 for (i=1; i<=AT[x][0]; i++)
   if (viz[AT[x][i]])
     DFST(AT[x][i]);
}

void DFS(int x)
{int i;
 viz[x]=1;
 for (i=1; i<=A[x][0]; i++)
   if (!viz[A[x][i]])
     DFS(A[x][i]);
 postordine[++nr]=x;
}

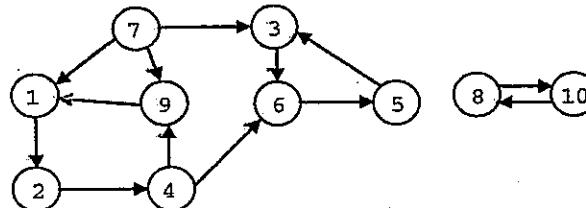

```

*Observație*

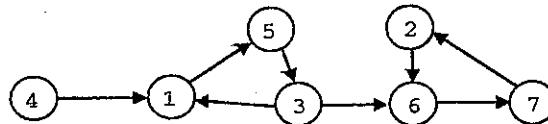
Pentru graful reprezentat prin liste de adiacență, complexitatea algoritmului Kosaraju-Sharir de descompunere în componente tare-conexe este de ordinul  $O(n+m)$ .

*Exerciții propuse*

1. Se consideră graful orientat din figura următoare. Descompuneți graful în componente tare-conexe.



2. Se consideră graful orientat cu  $n=23$  vârfuri și mulțimea arcelor:  $\{(1,5), (5,1), (6,1), (6,7), (7,8), (7,10), (10,8), (8,9), (8,11), (11,10), (9,11), (4,3), (22,4), (3,22), (3,19), (19,18), (18,3), (22,16), (22,17), (16,17), (20,16), (17,20), (20,21), (16,21), (12,2), (12,13), (13,2), (15,2), (14,2), (14,15)\}$ . Descompuneți graful în componente tare-conexe.
3. Construiți un graf tare-conex cu 7 vârfuri și cu număr minim de arce.
4. Se consideră graful orientat din figura următoare. Care este numărul minim de arce care trebuie să fie adăugate astfel încât graful să devină tare-conex?



5. Se consideră un graf orientat tare-conex. Stabiliți numărul circuitelor care conțin toate vâfurile grafului.
  - a. exact unul
  - b. cel puțin unul
  - c. nici unul
  - d. cel mult unul

(Bacalaureat, iulie 2003)
6. Se consideră un graf orientat cu 9 vârfuri astfel încât, pentru orice  $i$ , există câte un arc de la vârful  $i$  la vârful  $2 \cdot i$  și la vârful  $2 \cdot i + 1$  (pentru  $2 \cdot i \leq 9$ , respectiv  $2 \cdot i + 1 \leq 9$ ), precum și arcul  $(9,1)$ . Câte componente tare-conexe are acest graf?
7. Scrieți un program care să realizeze descompunerea în componente tare-conexe a unui graf orientat pe baza matricei închiderii tranzitive a grafului.
8. Scrieți un program care să construiască graful condensat al unui graf orientat. În graful condensat există un vârf pentru fiecare componentă tare-conexă a grafului dat. Există arc de la vârful  $x$  la vârful  $y$  în graful condensat dacă există cel puțin un drum de la un vârf situat în componentă tare-conexă corespunzătoare vârfului  $x$ , la un vârf situat în componentă tare-conexă corespunzătoare vârfului  $y$ .

## 1.8. Arbori

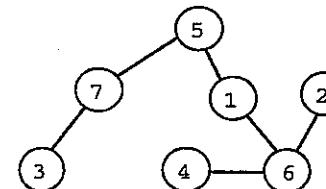
Un graf neorientat conex și aciclic se numește *arbore*.  
Un graf neorientat aciclic și neconex se numește *pădure*.

### Observație

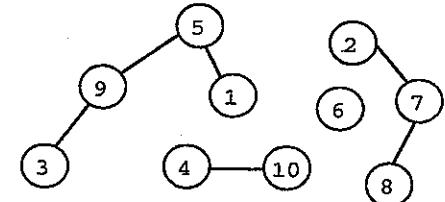
Fiecare componentă conexă a unui graf aciclic neconex (pădure) este un arbore.

### Exemple

Graful este un arbore, deoarece este conex și aciclic:



Graful este o pădure, deoarece este aciclic și neconex:



### Teoremă de caracterizare a arborilor

Fie  $G$  un graf neorientat cu  $n$  vârfuri. Următoarele afirmații sunt echivalente:

1.  $G$  este arbore.
2. Oricare două vârfuri din  $G$  sunt unite printr-un lanț simplu unic.
3.  $G$  este conex minimal (dacă suprimăm o muchie, graful obținut este neconex).
4.  $G$  este conex și are  $n-1$  muchii.
5.  $G$  este aciclic și are  $n-1$  muchii.
6.  $G$  este aciclic maximal (dacă adăugăm o muchie, graful obținut conține cicluri).

### Exercițiu propus

Demonstrați teorema de caracterizare a arborilor, considerând în ordine implicațiile  $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6 \Rightarrow 1$ .

### Determinarea unui arbore cu secvența gradelor dată

#### Teoremă

Numerele naturale  $0 < d_1 \leq d_2 \leq \dots \leq d_n$  ( $n \geq 2$ ) sunt gradele vâfurilor unui arbore dacă și numai dacă  $d_1 + d_2 + \dots + d_n = 2n - 2$ .

#### Demonstrație

Vom demonstra această teoremă, deoarece demonstrația oferă și o soluție constructivă a unui arbore cu secvența gradelor dată.

Condiția este necesară, deoarece orice arbore cu  $n$  vârfuri are  $n-1$  muchii, iar suma gradelor vâfurilor oricărui graf este dublul numărului de muchii. Deci  $d_1 + d_2 + \dots + d_n = 2n - 2$ .

Să demonstrăm că această condiție este suficientă. Fie  $0 < d_1 \leq d_2 \leq \dots \leq d_n$ , astfel încât  $d_1 + d_2 + \dots + d_n = 2n - 2$ . Demonstrăm inductiv că există un arbore cu gradele vâfurilor  $d_1, d_2, \dots, d_n$ .

Pentru  $n=1$ ,  $d_1=0$  (arborele format dintr-un vârf izolat), iar pentru  $n=2$ ,  $d_1+d_2=2$ , deci  $d_1=d_2=1$  (arborele format din două vârfuri adiacente).

Presupunem acum că proprietatea este adeverată pentru orice secvență de  $n$  numere naturale  $0 < d_1 \leq d_2 \leq \dots \leq d_n$ , astfel încât  $d_1+d_2+\dots+d_n=2n-2$  și demonstrăm că, pentru orice secvență  $0 < d'_1 \leq d'_2 \leq \dots \leq d'_{n+1}$ , astfel încât  $d'_1+d'_2+\dots+d'_{n+1}=2n$ , există un arbore cu  $n+1$  vârfuri cu secvența gradelor  $d'_1, d'_2, \dots, d'_{n+1}$ .

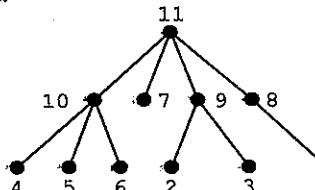
Observăm că există măcar un nod terminal  $x_1$  cu gradul  $d'_1=1$ ; altfel, dacă  $d_i \geq 2, \forall i \in \{1, 2, \dots, n+1\} \Rightarrow d'_1+d'_2+\dots+d'_{n+1} \geq 2(n+1)$ , ceea ce contrazice ipoteza. În mod analog observăm că există măcar un nod neterminant  $x_{n+1}$  cu gradul  $d'_{n+1}>1$  altfel încât, dacă  $d'_1=1, \forall i \in \{1, 2, \dots, n+1\} \Rightarrow d'_1+d'_2+\dots+d'_{n+1}=n+1 < 2n$ .

Să considerăm următoarea secvență de  $n$  numere întregi  $d'_2, \dots, d'_n, d'_{n+1}-1$ , cu proprietatea că  $d'_2+\dots+d'_n+d'_{n+1}=2n-2$ . Din ipoteza inducțivă, există un arbore  $A_n$  cu  $n$  vârfuri și secvența gradelor  $d'_2, \dots, d'_n, d'_{n+1}-1$ . Adăugăm la arborele  $A_n$  un vârf pe care îl unim printr-o muchie cu vârful având gradul  $d'_{n+1}-1$ . Obținem un arbore  $A_{n+1}$  cu gradele vârfurilor  $d'_1, d'_2, \dots, d'_{n+1}$ .

De exemplu, pentru  $n=11$  și sirul gradelor:

1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	2	3	4	4

obținem arborele din figură.



Vom reține gradele vârfurilor într-un vector  $d$ , ordonat crescător, cu suma componentelor egală cu  $2n-2$ . Pentru a reține arborele este suficient să reținem pentru fiecare vârf terminal „eliminat” din arbore și vârful neterminant cu care este adiacent. Spre deosebire de ideea din demonstrație, pentru a conserva ordinea în vectorul  $d$ , la fiecare pas al algoritmului, gradul care va fi decrementat va fi primul grad mai mare ca 1 și nu ultimul.

```

int s=0, i, VfT, VfNt;
for (i=1; i<=n; i++) s+=d[i];
if (s!=2*(n-1))
    fout<<"Suma gradelor trebuie sa fie 2(n-1)!\n";
else
    for (VfT=1; VfT<n; VfT++)
        //determin primul varf neterminant
        for (VfNt=VfT+1; VfNt<n && d[VfNt]==1; VfNt++)
            a[VfT]=VfNt;
            d[VfNt]--;
  
```

### Observație

Acest algoritm constructiv sugerează o metodă foarte eficientă de reprezentare a arborilor. Dacă  $A_n$  este un arbore cu vârfurile  $x_1, x_2, \dots, x_n$ , suprimăm vârful

terminal cu cel mai mic indice și muchia incidentă cu acesta și reținem  $a_1$ , vârful adiacent cu vârful terminal suprimat. Am obținut astfel un subgraf cu  $n-2$  muchii conex și aciclic, deci un arbore  $A_{n-1}$ . Repetăm procedeul pentru arborele  $A_{n-1}$ , determinând un alt doilea vârf,  $a_2$ , adiacent cu vârful terminal de indice minim ce va fi eliminat din  $A_{n-1}$  împreună cu muchia incidentă cu el și a.m.d., până când se obține un arbore  $A_2$  cu două vârfuri adiacente.

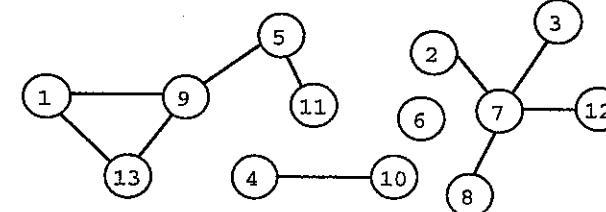
Am obținut astfel un sistem  $\{a_1, a_2, \dots, a_{n-2}\}$  de  $n-2$  numere ( $1 \leq a_i \leq n, \forall i \in \{1, 2, \dots, n-2\}$ ) asociat arborelui  $A_n$ , numit *codul Prüffer* al lui  $A_n$ .

Se poate demonstra că există o corespondență biunivocă între mulțimea arborilor  $A$  cu  $n$  vârfuri și mulțimea sistemelor  $\{a_1, a_2, \dots, a_{n-2}\}, a_i \in \{1, 2, \dots, n\}, \forall i \in \{1, 2, \dots, n-2\}$ .

Folosind acest rezultat, deducem că numărul arborilor ce se pot construi cu  $n$  vârfuri date este egal cu  $n^{n-2}$  (numărul funcțiilor definite pe o mulțime cu  $n-2$  elemente, cu valori într-o mulțime cu  $n$  elemente). Această formulă poartă numele de *formula lui Cayley*.

### Exerciții propuse

1. Se consideră graful neorientat din figura următoare. Care dintre componentele conexe ale grafului sunt arbori?



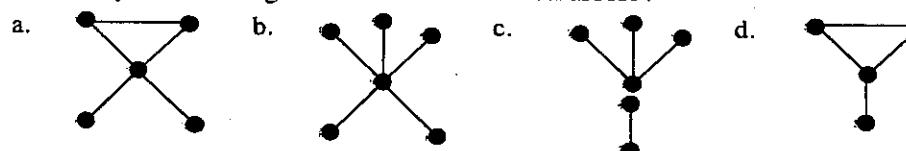
2. Se consideră următorul graf cu 7 vârfuri și 6 muchii, reprezentat prin lista muchiilor. Este acest graf un arbore?

1	1	2	4	5	4
3	7	5	5	6	6

3. Să considerăm un graf complet cu 6 vârfuri. Câte muchii trebuie să fie eliminate din graf pentru a obține un arbore?

4. Un graf neorientat are 20 de vârfuri, 30 de muchii și 5 componente conexe. Câte muchii trebuie să fie eliminate din graf pentru a obține o pădure?

5. Stabiliți care dintre grafurile următoare este un arbore:



(Simulare Bacalaureat, 2003)

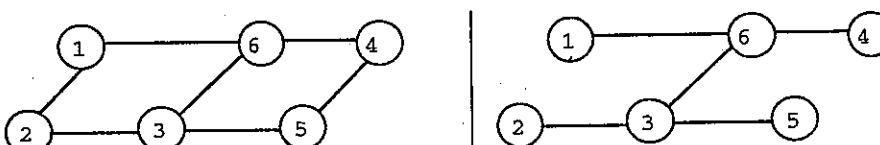
6. Memorarea unui arbore cu ajutorul matricei de adiacență este o metodă:  
 a. ineficientă      b. eficientă      c. recomandabilă      d. incorrectă  
 (Bacalaureat special, 2003)
7. Se consideră graful neorientat dat prin matricea de adiacență următoare. Stabiliti dacă se poate obține un arbore prin eliminarea uneia dintre muchiile grafului.
- |   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 |
- a. Da, prin eliminarea exact a unei muchii.  
 b. Da, prin eliminarea exact a două muchii.  
 c. Da, prin eliminarea exact a trei muchii.  
 d. Nu.  
 (Bacalaureat, iulie 2006)
8. Se dă următoarea secvență de numere naturale: 4, 3, 3, 1, 1, 1, 1, 1, 1. Să se verifice dacă această secvență poate fi secvența gradelor unui arbore. În caz afirmativ determinați un arbore cu această secvență a gradelor.
9. Se consideră următorul cod Prüffer: 3, 4, 4, 6, 6, 7, 9. Să se construiască arborele codificat.
10. Demonstrați că vârfurile oricărui arbore pot fi colorate cu două culori astfel încât oricare două vârfuri adiacente să fie colorate diferit.
11. Fie un graf neorientat reprezentat prin:  
 a. matrice de adiacență  
 b. liste de adiacență  
 c. lista muchiilor  
 Scrieți un program care să verifice dacă graful este arbore.
12. Scrieți un program care să genereze toți arborii cu  $n$  vârfuri.

## 1.9. Arbori parțiali

Un graf parțial conex și aciclic al unui graf neorientat se numește *arbore parțial*.

### Exemplu

Graful ilustrat în figura din dreapta este un arbore parțial al grafului din stânga.



### Teorema 1

Condiția necesară și suficientă ca un graf să conțină cel puțin un arbore parțial este ca graful să fie conex.

### Demonstratie

Necesitatea. Presupunem că graful admite un arbore parțial. Arborele parțial este conex și adăugând muchiile care sunt în graf, dar nu sunt în arborele parțial, el rămâne conex. Deci graful este conex.

Suficiența. Presupunem că graful este conex. Dacă graful este conex minimal, el este arborele parțial căutat. Altfel, există o muchie  $[x, y]$ , astfel încât graful parțial  $G_1$  obținut prin eliminarea muchiei  $[x, y]$  este conex. Dacă  $G_1$  este conex minimal, arborele parțial căutat este  $G_1$ , altfel continuăm procedeul de eliminare a muchiilor până când obținem un graf conex minimal, care va fi arborele parțial căutat.

### Teorema 2

Fie  $G$  un graf conex cu  $n$  vârfuri și  $m$  muchii. Numărul de muchii ce trebuie eliminate pentru a obține un arbore parțial este  $m-n+1$  (acesta se numește *numărul ciclomatic* al grafului).

### Demonstratie

Presupunem că prin eliminarea unui număr oarecare de muchii din  $G$  am obținut un graf  $G'$  fără cicluri (o pădure). Fiecare dintre componentele conexe ale lui  $G'$  este un arbore. Să notăm cu  $p$  numărul componentelor conexe, cu  $n_i$  numărul de vârfuri din componenta conexă  $i$ , unde  $i \in \{1, 2, \dots, p\}$  și cu  $m_i$  numărul de muchii din componenta conexă  $i$ ,  $i \in \{1, 2, \dots, p\}$ . Evident că  $m_i = n_i - 1$ ,  $\forall i \in \{1, 2, \dots, p\}$ .

Numărul de muchii din  $G'$  este  $(n_1 - 1) + (n_2 - 1) + \dots + (n_p - 1) = n - p$ . Deci au fost eliminate  $m - n + p$  muchii. Când  $G'$  este arbore, deci conex ( $p=1$ ), numărul muchiilor eliminate este  $m - n + 1$ .

### Arbore parțiali obținuți prin parcurgerea grafurilor

Prin parcurgerea unui graf neorientat conex fiecare vârf din graf va fi vizitat o singură dată. Exceptând vârful de start, pentru vizitarea fiecărui vârf se utilizează o singură muchie a grafului. În total pentru parcursul au fost utilizate  $n-1$  muchii (denumite *tree edges*) care nu formează cicluri, deci care constituie un arbore parțial al grafului dat.

### Observații

- În cazul în care graful parcurs nu este conex, parcursul se repetă pentru fiecare componentă conexă, obținându-se o pădure, formată din arborii parțiali corespunzători fiecărei componente conexe.
- Prin parcurgerea unui graf orientat se obține o arborescență (graf orientat fără circuite, în care fiecare vârf este accesibil din vârful de start).

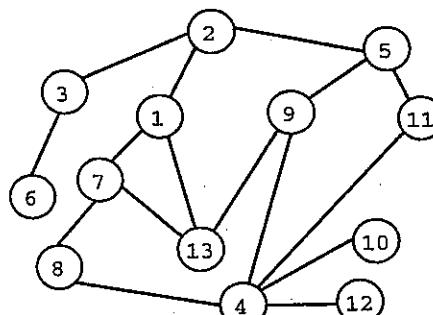
**Clasificarea muchiilor**

Prin parcurgere, muchiile grafului pot fi clasificate în 4 categorii:

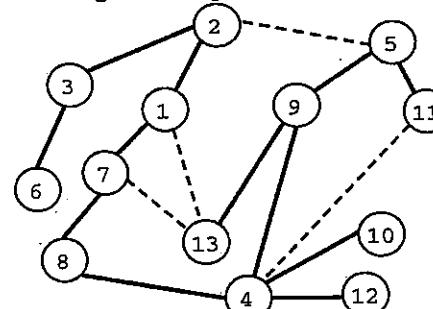
1. **Tree edges** – muchiile arborelui parțial; mai exact,  $[x, y]$  este **tree edge** dacă și numai dacă vârful  $y$  a fost vizitat explorând muchia  $[x, y]$ ;
2. **Back edges** – muchii care nu aparțin arborelui parțial și care conectează un vârf de un strămoș al său în arborele parțial (adică muchii de formă  $[x, y]$  care conectează pe  $x$  de un vârf  $y$  aflat în lanțul de la vârful de start la  $x$  mai aproape de vârful de start decât  $x$ );
3. **Forward edges** – muchii care nu aparțin arborelui parțial și care conectează un vârf de un descendant al său în arborele parțial (adică o muchie de formă  $[x, y]$  care conectează pe  $x$  de un vârf  $y$ , astfel încât  $x$  se află în lanțul de la vârful de start la  $y$  mai aproape de vârful de start decât  $y$ );
4. **Cross edges** – toate celelalte muchii.

**Exemplu**

Să considerăm următorul graf neorientat conex:



Să parcurgem în adâncime graful începând din vârful 2.



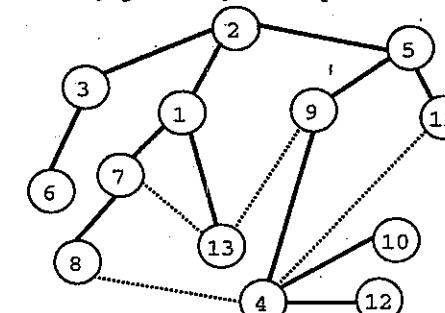
Am figurat cu linie continuă îngroșată muchiile arborelui parțial DFS (tree edges) și cu linie întreruptă muchiile de întoarcere (back edges).

**Observații**

1. Un graf (orientat sau neorientat) este aciclic dacă nu conține muchii de întoarcere (back edges).

2. Prin parcurgerea în adâncime a unui graf neorientat, muchiile pot fi clasificate doar în muchii ale arborelui parțial DFS (tree edges) sau muchii de întoarcere (back edges). Demonstrația o propunem ca exercițiu.

Să parcurgem acum același graf în lățime începând cu vârful 2.



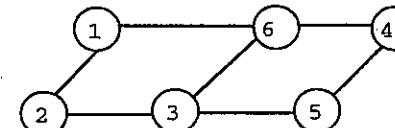
Am figurat cu linie continuă îngroșată muchiile arborelui parțial BFS (tree edges) și cu linie punctată celelalte muchii (cross-edges).

**Observații**

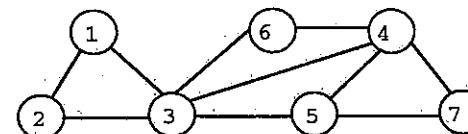
1. Pentru orice vârf  $x$  din graf, lanțul unic care unește vârful de start de  $x$  în arborele parțial BFS reprezintă un lanț cu număr minim de muchii de la vârful de start la  $x$  în graf. Lungimea acestui lanț o vom numi distanță de la vârful de start la  $x$  și o vom nota  $d[x]$ .
2. Prin parcurgerea BFS a unui graf neorientat muchiile pot fi clasificate doar în două categorii: muchii ale arborelui parțial BFS (tree edges) și cross edges. Dacă  $[x, y]$  este **tree edge**, atunci  $d[y] = d[x] + 1$ . Dacă  $[x, y]$  este **cross edges**, atunci  $d[x] = d[y]$  sau  $d[y] = d[x] + 1$ .

**Exerciții propuse**

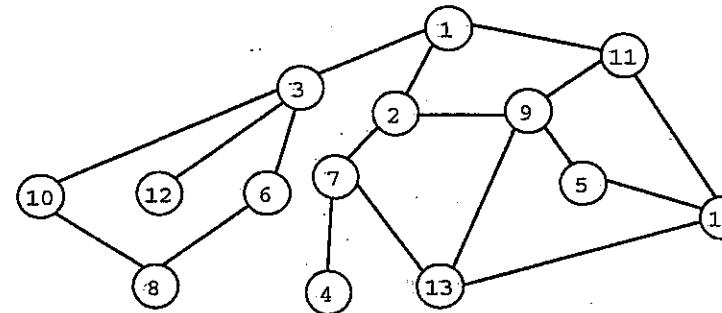
1. Se consideră graful din figura următoare. Determinați toți arborii parțiali ai acestui graf.



2. Se consideră graful din figura următoare. Prin eliminarea muchiilor  $[2, 3]$ ,  $[3, 5]$ ,  $[4, 5]$  și  $[4, 7]$  se obține un arbore parțial al acestui graf?



3. Se consideră graful din figura următoare. Construiți arborele parțial DFS și arborele parțial BFS, considerând ca vârf de start vârful 1.



4. Fie  $G$  un graf orientat și  $x, y$  două vîrfuri din graf astfel încât vîrful  $x$  a fost vizitat înaintea vîrfului  $y$  în timpul parcurgerii DFS. Determinați valoarea de adevăr a următoarei propoziții: „vîrful  $x$  este un strămoș al vîrfului  $y$  în arborele parțial DFS” (adică vîrful  $x$  se află pe drumul unic de la vîrful de start la vîrful  $y$ , mai aproape de vîrful de start decât  $y$ ). Demonstrați faptul că propoziția este adevărată sau oferiți un contraexemplu în caz contrar.
5. Scrieți un program care să genereze toți arborii parțiali ai unui graf neorientat.
6. Scrieți un program care să determine arborele parțial DFS și arborele parțial BFS pentru un graf neorientat conex, vîrful de start fiind specificat.
7. Scrieți un program care să clasifice muchiile unui graf neorientat/orientat (*tree edge, back edge, forward edge, cross edge*) în urma unei parcurgeri în adâncime și în urma unei parcurgeri în lățime.

## 1.10. Arbori parțiali de cost minim

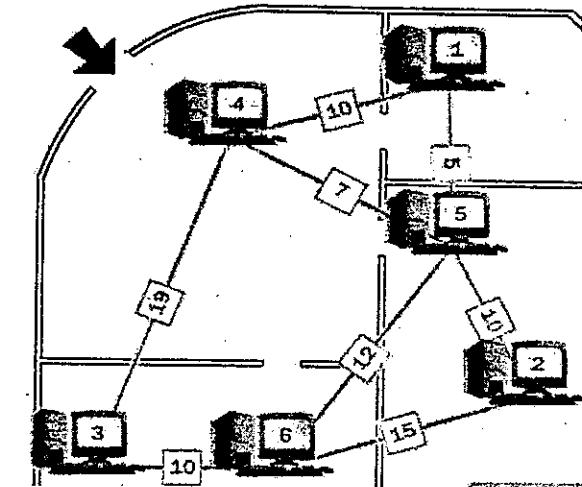
De cele mai multe ori, în practică nu interesează generarea tuturor arborilor parțiali ai unui graf conex, ci numai a celor care satisfac anumite condiții de optim. De exemplu, să presupunem că dorim să proiectăm rețeaua intranet a unei firme. Cunoaștem poziția calculatoarelor în sediul firmei, care sunt perechile de calculatoare între care s-ar putea trage cablu, precum și lungimea cablului necesar. Scopul nostru este să conectăm calculatoarele astfel încât oricare două calculatoare să poată comunica, iar lungimea totală a cablului consumat să fie minimă.

În primul rând observăm că putem asocia problemei un graf: vîrfurile sunt asociate calculatoarelor din firmă; dacă între două calculatoare putem trage cablu, considerăm că există muchie în graf între vîrfurile asociate calculatoarelor respective. Mai mult, observăm că graful este ponderat ( fiecare muchie are asociat un cost – lungimea cablului necesar).

Să demonstrăm, în primul rând, că soluția optimă de interconectare a calculatoarelor corespunde unui arbore parțial în graful asociat. În primul rând pentru ca oricare două calculatoare să poată comunica, graful parțial trebuie să fie conex. Pentru a obține costul minim, graful parțial trebuie să fie conex minimal (în caz contrar, prin eliminarea unei muchii, graful parțial ar rămâne conex, iar costul său

total ar fi mai mic). Conform teoremei de caracterizare a arborilor, un graf conex minimal este arbore.

De exemplu, pentru următoarea aranjare a calculatoarelor :



o soluție optimă ar fi conectarea calculatoarelor [1,5], [4,5], [2,5], [3,6] și [5,6]. În acest caz soluția este unică, dar, în general, există mai multe soluții.

### Enunțul problemei

Se consideră  $G=(V, E)$  un graf neorientat conex și  $c:E \rightarrow \mathbb{R}$  o funcție prin care se asociază fiecărei muchii din graf un număr real denumit costul muchiei. Să se determine un arbore parțial de cost minim.

Costul unui arbore este egal cu suma costurilor muchiilor arborelui.

Vom prezenta doi algoritmi fundamentali pentru rezolvarea acestei probleme: algoritmul lui Kruskal și algoritmul lui Prim. Ambii algoritmi se bazează pe o strategie *Greedy*, dar maniera de abordare este diferită.

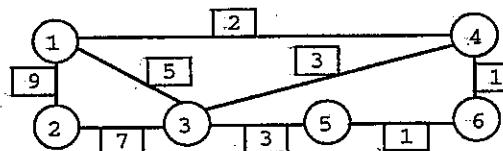
### Algoritmul lui Kruskal

Pentru a determina un arbore parțial de cost minim se consideră inițial că nici o muchie nu este selectată. Cu alte cuvinte, se pleacă de la o pădure formată din  $n$  arbori (unde  $n$  reprezintă numărul de vîrfuri din graf), fiecare arbore fiind format dintr-un singur vîrf.

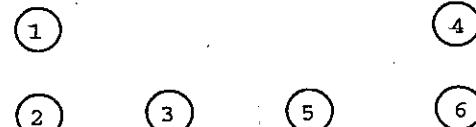
La fiecare pas al algoritmului se selecteză o muchie de cost minim care nu a mai fost selectată și care nu formează cicluri cu cele deja selectate.

**Exemplu**

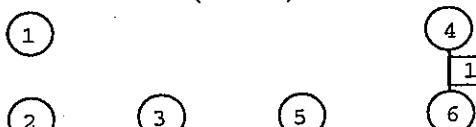
Să considerăm următorul graf ponderat și să aplicăm pas cu pas algoritmul lui Kruskal pentru determinarea unui arbore parțial de cost minim al acestui graf.



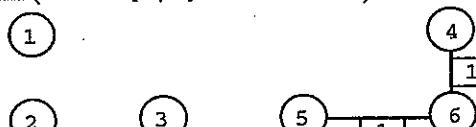
Înțial fiecare vârf este izolat:



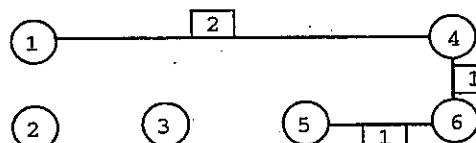
Selectăm o muchie de cost minim (costul 1):



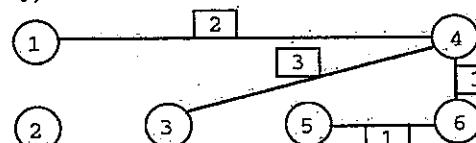
Observați că prin selectarea muchiei  $[4, 6]$  două componente conexe s-au unificat (vârfurile 4 și 6 fac parte acum din aceeași componentă conexă). Să selectăm încă o muchie de cost minim (muchia  $[5, 6]$  având costul 1):



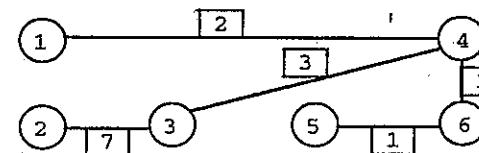
Prin selectarea muchiei  $[5, 6]$  s-au unificat componentele conexe ale extremităților ei (vârfurile 4, 5 și 6 formează acum o singură componentă conexă). Selectăm următoarea muchie de cost minim (muchia  $[1, 4]$  de cost 2):



Graful are acum trei componente conexe. Selectăm următoarea muchie de cost minim, muchia  $[3, 4]$ , de cost 3.



Următoarea muchie de cost minim ar fi muchia  $[3, 5]$ . Dar prin selectarea acestei muchii s-ar forma un ciclu (deoarece extremitățile muchiei, vârfurile 3 și 5, sunt în aceeași componentă conexă). Prin urmare, nu vom selecta muchia  $[3, 5]$ , ci vom analiza următoarea muchie de cost minim, muchia  $[1, 3]$ . Si această muchie ar forma un ciclu dacă ar fi selectată, prin urmare o ignorăm și selectăm următoarea muchie de cost minim, muchia  $[2, 3]$ , de cost 7.



În acest moment sunt selectate 5 muchii care nu formează cicluri, deci am obținut un arbore parțial. În plus, acest arbore este de cost minim.

*Implementarea algoritmului lui Kruskal*

Pentru implementarea algoritmului este necesară rezolvarea următoarelor două probleme: cum extragem muchia de cost minim și cum testăm dacă muchia selectată formează sau nu cicluri cu cele deja selectate.

Pentru a extrage minimul, o primă idee ar fi să sortăm muchiile crescător după cost și să parcurgem secvențial muchiile ordonate. În cazul în care arborele parțial de cost minim este găsit suficient de repede, un număr mare de muchii rămân netestate și în acest caz s-ar pierde timp inutil cu sortarea acestor muchii. O altă idee, mai eficientă, ar fi să organizăm muchiile grafului ca un *min-heap*, structură pe care o vom studia ulterior și care permite extragerea eficientă a minimului.

Pentru a testa dacă o muchie formează cicluri cu muchiile deja selectate este suficient să testăm dacă extremitățile muchiei se găsesc în aceeași componentă conexă. Pentru aceasta va trebui să ținem permanent evidența componentelor conexe (arborilor) care se formează.

*Reprezentarea informațiilor*

1. Vom reprezenta graful prin lista muchiilor.
2. Arborele parțial de cost minim se va memora într-un vector A cu  $n-1$  componente, în care reținem indicii din graf ai muchiilor selectate.
3. Evidența componentelor conexe o vom reține cu ajutorul unui vector c cu n componente,  $c[i] =$  componentă conexă căreia îi aparține vârful i. Componentele conexe vor fi identificate printr-un reprezentant (vârful cu indicele cel mai mic din componentă conexă respectivă).

```
#include <fstream.h>
#define NMaxVf 50
#define NMaxMuchii NMaxVf * (NMaxVf - 1) / 2

struct Muchie {int e1, e2, cost; };

Muchie G[NMaxMuchii];
int A[NMaxVf], c[NMaxVf];
int n, m;
```

```

void Initialize()
{
    int i;
    ifstream fin("Kruskal.in");
    fin>>n>>m;
    for (i=1; i<=m; i++)
        fin>>G[i].e1>>G[i].e2>>G[i].cost;
    for (i=1; i<=n; i++) c[i]=i;
    fin.close();
}

void Afisare()
{
    int i, CostAPM=0;
    cout<<"Arborele parțial de cost minim este :\n";
    for (i=1; i<n; i++)
        {cout<<"[<<G[A[i]].e1<<,"<<G[A[i]].e2<<]" cost=";
         <<G[A[i]].cost<<endl;
         CostAPM+=G[A[i]].cost;}
    cout<<"Costul APM= "<<CostAPM<<endl;
}

void SortareMuchii(int st, int dr)
{
    int i, j;
    Muchie x;
    if (st<dr)
        {x=G[st]; i=st; j=dr;
         while (i<j)
             {while (i<j&&G[j].cost>=x.cost) j--;
              G[i]=G[j];
              while (i<j&&G[i].cost<=x.cost) i++;
              G[j]=G[i];
            }
         G[i]=x;
         SortareMuchii(st, i-1);
         SortareMuchii(i+1, dr);
        }
}

int main()
{
    int i, j, min, max, NrMSel;
    Initialize();
    SortareMuchii(1, m);
    NrMSel=0; // numarul de muchii selectate
    for (i=1; NrMSel<n-1; i++)
        if (c[G[i].e1]!=c[G[i].e2])
            // muchia i nu formeaza cicluri cu cele deja selectate
            { // selectez muchia i
                A[++NrMSel]=i;
                // unific componentele conexe ale extremitatilor
                if (c[G[i].e1]<c[G[i].e2])
                    {min=c[G[i].e1];
                     max=c[G[i].e2]; }
            }
}

```

```

    else
        {min=c[G[i].e2];
         max=c[G[i].e1]; }
    for (j=1; j<=n; j++)
        if (c[j]==max) c[j]=min;
    }
    Afisare();
    return 0;
}

```

### Teoremă

Algoritmul lui Kruskal generează un arbore parțial de cost minim.

### Demonstratie

1. Algoritmul selectează numărul maxim de muchii care nu formează cicluri, deci, conform teoremei de caracterizare a arborilor, se obține un arbore parțial.
2. Să demonstrăm că arborele parțial obținut în urma aplicării algoritmului lui Kruskal este un arbore parțial de cost minim.

Fie  $A = (a_1, a_2, \dots, a_{n-1})$  muchiile arborelui rezultat în urma aplicării algoritmului, în ordinea selectării lor.

Presupunem prin reducere la absurd că arborele obținut nu este de cost minim, deci există  $A' = (a'_1, a'_2, \dots, a'_{n-1})$  un alt arbore parțial, astfel încât  $c(A') < c(A)$ .

Fie  $k = \min\{i \mid 1 \leq i \leq n, a_i \neq a'_i\}$ , primul indice de la care  $A$  și  $A'$  diferă. Evident  $c(a_i) \leq c(a'_i)$ ,  $\forall j \in \{1, \dots, n-1\}$ , altfel algoritmul ar fi selectat muchia  $a_j$  în loc de  $a_i$ , deoarece ar fi avut cost mai mic și nu formează cicluri cu  $a_1, \dots, a_{i-1}$ . Adaug la  $A'$  muchia  $a_i$ . Se formează un ciclu în care intervin doar muchii din multimea  $\{a'_1, \dots, a'_{n-1}\}$ . Elimin una dintre muchiile ciclului diferită de  $a_i$ . Se obține un arbore  $A'' = (a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1})$  care are  $i$  muchii comune cu  $A$ . În plus,  $c(A'') - c(A') = c(a_i) - c(a'_i) \leq 0 \Rightarrow c(A'') \leq c(A')$ .

Repetăm procedeul de înlocuire a muchiilor din  $A'$  cu muchiile din  $A$ . Obținem  $c(A') \geq c(A'') \geq \dots \geq c(A)$ . Dar am presupus că  $c(A') < c(A) \Rightarrow$  contradicție. Deci  $A$  este un arbore parțial de cost minim.

### Complexitatea algoritmului

Sortarea muchiilor prin algoritmul de sortare rapidă are complexitatea, în medie,  $O(m \log n)$ . Algoritmul cercetează în cel mai defavorabil caz toate cele  $m$  muchii pentru a selecta  $n-1$  care nu formează cicluri. Selectarea unei muchii implică și o operație de unificare a două componente conexe care se execută în  $O(n)$  în această implementare. Deci, per total, complexitatea este  $O(m \log m + n^2)$ .

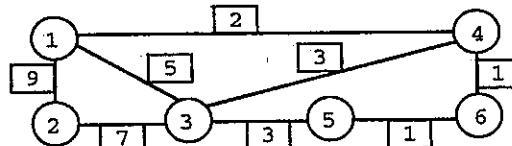
### Algoritmul lui Prim

Ca și algoritmul lui Kruskal, algoritmul lui Prim utilizează o strategie *Greedy*. Spre deosebire de algoritmul lui Kruskal care la fiecare pas selectă o muchie, algoritmul lui Prim selectează la fiecare pas un singur vârf și o muchie incidentă cu acesta.

Inițial se pleacă de la un arbore format dintr-un singur vârf. La fiecare pas se selectează o muchie de cost minim astfel încât subgraful parțial generat de mulțimea vârfurilor selectate și mulțimea muchiilor selectate odată cu acestea să formeze un arbore.

#### Exemplu

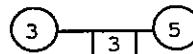
Să aplicăm pas cu pas algoritmul lui Prim pentru graful din figura următoare, considerând ca vârf de start vârful 3.



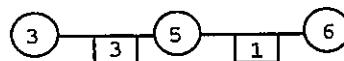
Initial :

3

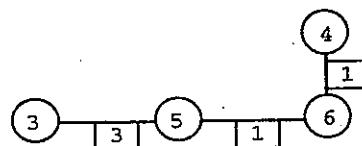
Selectăm un vârf care este incident cu o muchie de cost minim, în cazul acesta muchia incidentă cu vârful 3 (vârful 5 și muchia [3,5]):



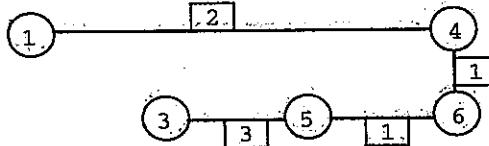
Selectăm un vârf care este incident cu o muchie de cost minim, muchie incidentă cu unul dintre vârfurile deja selectate (vârful 6 și muchia [5,6]):



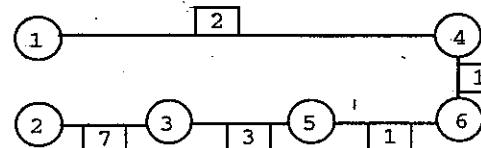
Selectăm din nou un vârf incident cu o muchie de cost minim, incidentă cu unul dintre vârfurile subgrafului construit la pasul anterior (vârful 4 și muchia [4,6]):



Selectând încă un vârf incident cu o muchie de cost minim, muchie care are o extremitate în mulțimea vârfurilor deja selectate (vârful 1 și muchia [1,4]) obținem:



În sfârșit, selectăm vârful 2 și muchia [2,3] (muchia de cost minim care are o extremitate 2 și cealaltă extremitate în mulțimea vârfurilor deja selectate):



Costul arborelui obținut este  $1+1+3+2+7=14$  (evident, același cu costul arborelui parțial obținut prin algoritmul lui Kruskal).

Observați că la fiecare pas se selectează un nou vârf, adjacent cu unul dintre vârfurile subgrafului deja construit, astfel încât muchia corespunzătoare să fie de cost minim. Nodul nou adăugat va fi terminal și deci nu se vor obține cicluri, iar subgraful construit este la fiecare pas conex, deci arbore.

#### Implementarea algoritmului lui Prim

##### Reprezentarea informațiilor

1. Reprezentăm graful prin matricea costurilor c.
2. z va fi mulțimea vârfurilor neselectate în subgraf reprezentată prin vector caracteristic ( $z[i]=1$  dacă i este vârf neselectat și 0 în caz contrar).
3. Vom utiliza un vector  $c_{min}$ , cu n componente, în care pentru fiecare vârf  $x$  din z vom reține costul minim al muchiilor ce unesc vârful  $x$  cu un vârf  $v$  din subgraf:  $c_{min}[x]=\min\{c([x,v]) \mid v \in x \setminus z\}$ . Vom denumi  $c_{min}[x]$  cheia vârfului  $x$ .
4. Reținem arborele parțial de cost minim, memorând vârfurile grafului în ordinea în care au fost atinse într-un vector p cu n componente;  $p[x]=$  vârful din care a fost atins  $x$ .

```
include <iostream.h>
#define NMaxVf 100
#define Inf 10000

int n, r, i, VfMin, nrV;
double G[NMaxVf][NMaxVf];
int p[NMaxVf], Z[NMaxVf];
double cmin[NMaxVf], CostMin;

void Initializare()
{
    int i, j, k;
    double c;
    ifstream fin("prim.in");
    fin>>n>>r;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++) G[i][j]=Inf;
```

```

for (i=1; i<=n; i++)
{ G[i][i]=0;
  fin>>nrv; // nr. de varfururi adiacente cu i
  for (j=1; j<=nrv; j++)
    { /* citesc varfurile adiacente cu i
       și costurile muchiilor corespunzătoare */
      fin>>k>>c;
      G[i][k]=c; }
}
for (i=1; i<=n; i++)
{ cmin[i]=G[r][i];
  p[i]=r; Z[i]=1; }
Z[r]=0; p[r]=0; nr=n-1;
fin.close();
}

void AfisareAPM()
{ int i;
  double cost=0;
  cout<<"Muchiile APM sunt: ";
  for (i=1; i<=n; i++)
    if (i!=r)
      { cout<<'['<<p[i]<<','<<i<<"] ";
        cost+=G[i][p[i]]; }
  cout<<endl<<"Costul APM "<<cost<<endl;
}

int main()
{
  Initializare();
  // cat timp mai există varfururi neselectate
  while (nrv)
  { // determin nodul din Z de cheie min
    CostMin=Inf;
    for (i=1; i<=n; i++)
      if (Z[i] && CostMin>cmin[i])
        { CostMin=cmin[i];
          VfMin=i; }
    // elimin VfMin din multimea varfurilor neselectate
    Z[VfMin]=0;
    nrV--;
    // actualizez cheile varfurilor din Z
    for (i=1; i<=n; i++)
      if (Z[i] && G[i][VfMin]<cmin[i])
        { p[i]=VfMin;
          cmin[i]=G[i][VfMin]; }
  }
  AfisareAPM();
  return 0;
}

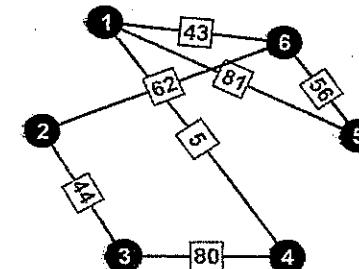
```

### Complexitatea algoritmului

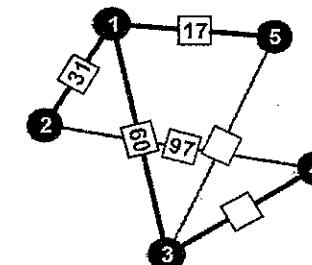
Algoritmul execută  $n-1$  pași, la fiecare pas selectând un vârf din graf de cheie minimă și reactualizând cheile vârfurilor neselectate, operație de complexitate  $O(n)$ . Deci algoritmul are complexitatea timp  $O(n^2)$ .

### Exerciții propuse

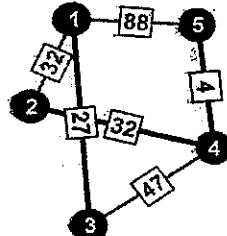
- Dați cel puțin două exemple de situații practice, întâlnite în viața cotidiană, în care se impune determinarea unui arbore parțial de cost minim.
- Se consideră graful din figura următoare. Determinați un arbore parțial de cost minim al acestui graf,
  - aplicând pas cu pas algoritmul lui Kruskal;
  - aplicând pas cu pas algoritmul lui Prim.



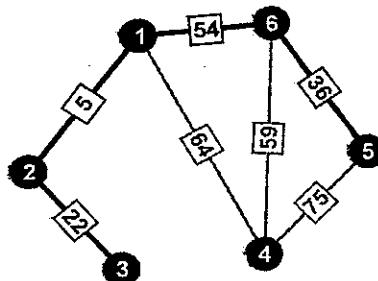
- Să considerăm graful din figura următoare, în care costurile unor muchii nu sunt specificate. Completăți costurile lipsă astfel încât muchiile marcate cu linie îngroșată să reprezinte muchiile unui arbore parțial de cost minim.



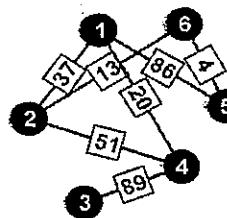
- Se consideră graful din figura următoare, în care cu linie îngroșată sunt ilustrate muchiile selectate după primii trei pași din algoritmul lui Kruskal. Ce muchie va fi selectată la pasul următor?



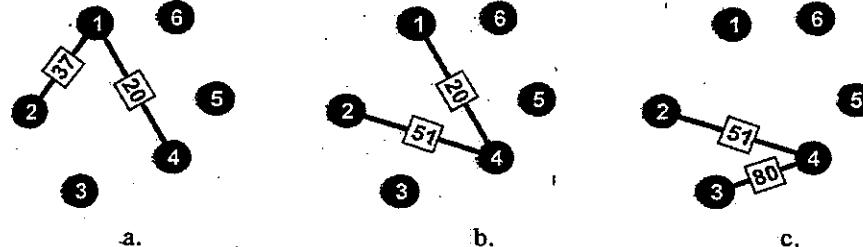
5. Se consideră graful din figura următoare, în care cu linie îngroșată sunt ilustrate muchiile care au fost selectate după aplicarea a 4 pași din algoritmul lui Kruskal. Care este în acest moment conținutul vectorului  $c$ , în care este memorată evidența componentelor conexe?



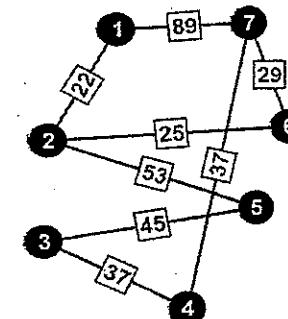
6. Determinați valoarea de adevăr a următoarei propoziții: *La fiecare pas în algoritmul lui Kruskal graful parțial obținut este conex.*
7. Explicați cum ați putea aplica algoritmul lui Prim pentru a obține un arbore parțial de cost maxim?
8. Se consideră graful din figura următoare:



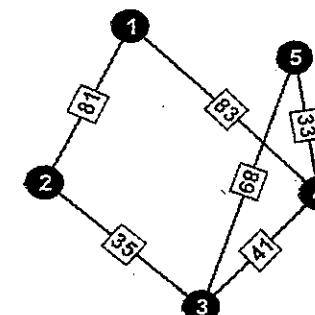
Vom considera ca vârf de start vârful 1. Care dintre următoarele grafuri parțiale se obțin după executarea a doi pași din algoritmul lui Prim?



9. Se consideră graful din figura următoare și vârful de start 1. După aplicarea a 4 pași din algoritmul lui Prim, valorile variabilelor  $z$ ,  $c_{min}$  și  $p$  sunt:  
 $z: \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0$   
 $c_{min}: \quad 0 \quad 22 \quad 37 \quad 37 \quad 53 \quad 25 \quad 29$   
 $p: \quad 0 \quad 1 \quad 0 \quad 7 \quad 0 \quad 2 \quad 6$   
Care va fi următorul vârf selectat?



10. Care va fi conținutul vectorului  $c_{min}$  după selectarea vârfurilor 1 și 2 din graful din figura următoare, utilizând algoritmul lui Prim?

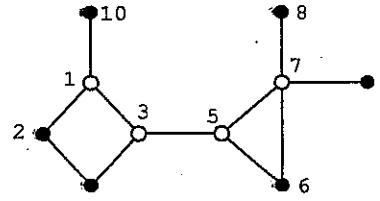


11. Explicați cum ați putea aplica algoritmul lui Prim pentru a obține un arbore parțial pentru care produsul costurilor muchiilor este minim. Presupunem că toate costurile muchiilor sunt numere reale strict pozitive.

## 1.11. Biconexitate

Fie  $G = (V, E)$  un graf neorientat conex. Vârful  $v \in X$  se numește punct de articulație dacă subgraful obținut prin eliminarea vârfului  $v$  și a muchiilor incidente cu acesta nu mai este conex.

De exemplu, pentru graful din figură, punctele de articulație sunt 1, 3, 5, 7.

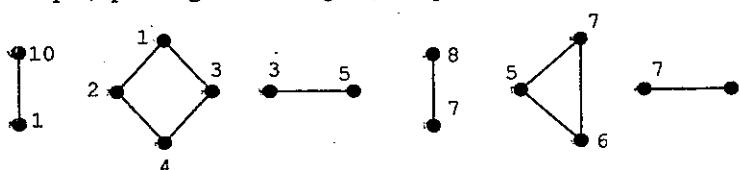


Un graf se numește biconex dacă nu are puncte de articulație.

În multe aplicații practice care se pot modela cu ajutorul grafurilor nu sunt de dorit punctele de articulație. De exemplu, într-o rețea de telecomunicații, dacă o centrală dintr-un punct de articulație se defectează rezultatul este nu doar întreruperea comunicării cu centrala respectivă, ci și cu alte centrale.

O componentă biconexă a unui graf este un subgraf biconex maximal cu această proprietate.

De exemplu, pentru graful din figură, componentele biconexe sunt:



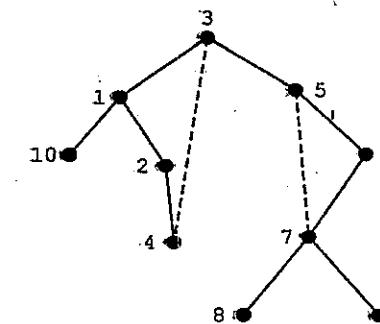
### Observații

1. Componentele biconexe ale unui graf reprezintă o parte a mulțimii muchiilor grafului.
2. Punctele de articulație aparțin la cel puțin două componente biconexe.

Pentru a descompune graful în componente biconexe vom utiliza proprietățile parcurgerii DFS. Parcugând graful DFS, putem clasifica muchiile grafului în:

- muchii care aparțin arborelui parțial DFS (*tree edges*);
- muchii  $[u, v]$  care nu aparțin arborelui și care unesc vârful  $u$  cu un strămoș al său  $v$  în arborele parțial DFS numite muchii de întoarcere (*back edges*). Acestea sunt marcate în exemplu punctat.

De exemplu, graful precedent poate fi redesenat, clasificând muchiile în raport cu arborele parțial DFS cu rădăcina 3:



Observăm că rădăcina arborelui parțial DFS este punct de articulație dacă și numai dacă are cel puțin doi descendenți, între vârfuri din subarbore diferenți ai rădăcinii neexistând muchii. Mai mult, un vârf  $x$  oarecare nu este punct de articulație dacă și numai dacă din orice fiu  $y$  al lui  $x$  poate fi atins un strămoș al lui  $x$  pe un lanț format din descendenți ai lui  $x$  și o muchie de întoarcere (un drum „de siguranță” între  $x$  și  $y$ ).

Pentru fiecare vârf  $x$  al grafului definim  $dfn(x)$ =numărul de ordine al vârfului  $x$  în parcurgerea DFS a grafului (*depth-first-number*).

De exemplu:

$x$	1	2	3	4	5	6	7	8	9	10
$dfn(x)$	2	4	1	5	6	7	8	9	10	3

Observăm că dacă  $x$  este un strămoș al lui  $y$  în arborele parțial DFS atunci  $dfn(x) < dfn(y)$ .

Pentru fiecare vârf  $x$  din graf definim  $low(x)$ =numărul de ordine al primului vârf din parcurgerea DFS ce poate fi atins din  $x$  pe un alt lanț decât lanțul unic din arborele parțial DFS.

$low(x) = \min\{dfn(x), \min\{low(y) | y \text{ fiu al lui } x, \min\{dfn(y) | [x, y] \text{ muchie de întoarcere}\}\}$ .

De exemplu:

$x$	1	2	3	4	5	6	7	8	9	10
$dfn(x)$	2	4	1	5	6	7	8	9	10	3
$low(x)$	1	1	1	1	6	6	6	9	10	3

Caracterizăm punctele de articulație dintr-un graf astfel:  $x$  este punct de articulație dacă și numai dacă este rădăcina unui arbore parțial DFS cu cel puțin doi descendenți sau, dacă nu este rădăcină, are un fiu  $y$  astfel încât  $low(y) \geq dfn(x)$ .

Pentru exemplul din figură, nodul 3 este punct de articulație deoarece este rădăcina arborelui parțial DFS și are doi descendenți, nodul 7 este punct de articulație, deoarece  $low(8) = 9 \geq dfn(7) = 8$ , nodul 5 este punct de articulație, deoarece  $low(6) = 6 \geq dfn(5) = 6$ , iar nodul 1 este punct de articulație, deoarece  $low(10) = 3 \geq dfn(1) = 2$ .

### Reprezentarea informațiilor

1. Vom reprezenta graful prin liste de adiacență alocate static.
2. Vectorii `dfn[]` și `low[]` conțin valorile calculate în timpul parcurgerii DFS.
3. Variabila globală `num` este utilizată pentru a calcula numărul de ordine a vârfului curent în parcurgerea în adâncime.
4. Variabila globală `nrfii` reține numărul fiilor vârfului de start în arborele parțial DFS.
5. Vom utiliza o stivă `S` în care vom reține muchiile din graf (atât pe cele care aparțin arborelui să și cele de întoarcere), în ordinea în care sunt întâlnite în timpul parcurgerii. Atunci când identificăm o componentă biconexă, mai exact, când identificăm un nod `u` care are un fiu `x` astfel încât `low(x) ≥ dfn(u)`, eliminăm din stivă toate muchiile din componentă biconexă respectivă. Funcția `Initializare()` initializează stiva cu o muchie fictivă (cu extremitatea finală egală cu vârful de start și extremitatea inițială -1, un nod inexistent).
6. Vom reprezenta mulțimea punctelor de articulație identificate în timpul parcurgerii în adâncime prin vectorul caracteristic `Art` (`Art[i]=1`, dacă `i` este punct de articulație și 0, în caz contrar).

În funcția `main()`, după citire și initializare, vom apela funcția `Biconex(start, -1)` care realizează descompunerea grafului în componente biconexe, apoi afișăm punctele de articulație.

```
#include <stdio.h>
#define NMax 101
#define MMax NMax*(NMax-1)/2

int n, num, Vfs, nrfii, nr, start=3;
int A[NMax][NMax];
int dfn[NMax], low[NMax], Art[NMax];

struct Muchie {int f, t;} S[MMax];
void Citire(void);
void Biconex(int, int);
void Initializare(void);
void Afisare_Comp_Biconexa(int, int);

int main()
{
    int i;
    Citire();
    Initializare();
    Biconex(start, -1);
    //Afisez punctele de articulație
    if (nrfii>1) //start este punct de articulație
        Art[start]=1;
    printf("Punctele de articulație sunt: ");
    for (i=1; i<=n; i++)
        if (Art[i]) printf("%d ", i);
    return 0;
}
```

```
void Citire()
{
    FILE * fin=fopen("graf.in", "r");
    int x, y, m, i;
    fscanf(fin, "%d %d", &n, &m);
    for (i=0; i<m; i++)
        fscanf(fin, "%d %d", &x, &y);
    A[x][0]++; A[x][A[x][0]]=y;
    A[y][0]++; A[y][A[y][0]]=x;
}

void Initializare()
{
    int i;
    for (i=1; i<=n; i++) dfn[i]=low[i]=-1;
    S[0].f=start; S[0].t=-1; Vfs=0;
}

int min(int x, int y)
{
    if (x<y) return x;
    return y;
}

void Biconex(int u, int tu)
//u este nodul curent; tu este nodul parinte al lui u
{
    int x, p;
    dfn[u]=low[u]=++num;
    //parcurg lista de adiacenta a nodului u
    for (p=1; p<=A[u][0]; p++)
        if (x=A[u][p]); //x este un nod adjacent cu u
            if (x!=tu && dfn[x]<dfn[u])
                //insereză în stiva S muchia {u,x}
                {S[++Vfs].f=x; S[Vfs].t=u; }
            if (dfn[x]==-1) //x nu a mai fost vizitat
                if (u==start) //am gasit un fiu al varfului start
                    nrfii++;
                Biconex(x, u);
                low[u]=min(low[u], low[x]);
            if (low[x]>=dfn[u]) //u este punct de articulație
                //am identificat o componentă biconexă,
                //formată din muchiile din stiva S pana la
                //întâlnirea muchiei {u,x}
                if (u!=start) Art[u]=1;
                Afisare_Comp_Biconexa(x, u);
            }
        else //x a mai fost vizitat
            if (x!=tu)
                //x nu este tatal lui u,
                //deci {u,x} e muchie de întoarcere la u la x
                low[u]=min(low[u], dfn[x]);
}
}
```

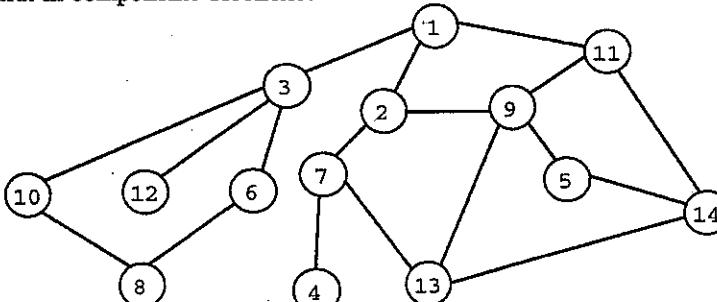
```

void Afisare_Comp_Biconexa(int x, int u)
//afiseaza componenta biconexa a muchiei [x,u]
{Muchie p;
 nr++;           //incrementez numarul de componente biconexe
printf("Componenta biconexa %d contine muchiile:\n",nr)
do
    {p=S[Vfs--]; //extrag o muchie din stiva
     printf("%d %d\n", p.t, p.f); }
while (p.t!=u || p.f!=x);
}

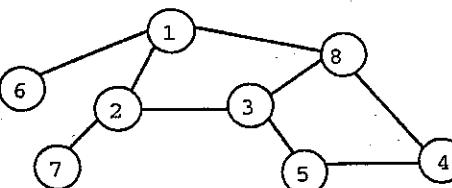
```

### *Exercitii propuse*

1. Care sunt punctele de articulație ale grafului din figura următoare? Descompuneți graful în componente biconexe.



2. Adăugați un număr minim de muchii în graful următor, astfel încât să obțineți un graf biconex.



3. Se numește *punte* o componentă biconexă formată dintr-o singură muchie. Scrieți un program care să identifice punctile unui graf.

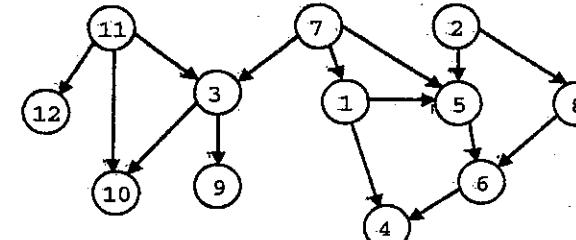
### 1.12. Descompunere pe niveluri a unui graf fără circuite

Fie  $G = (V, E)$  un graf orientat fără circuite. A descompune graful pe niveluri înseamnă a determina o partitie a multimii vârfurilor grafului cu următoarele proprietăți:

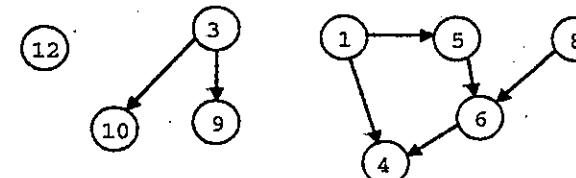
- multimile partiei sunt denumite niveluri ; nivelurile sunt numerotate de la 1 ;
  - dacă există arc de la vîrful  $x$  la vîrful  $y$ , atunci vîrful  $x$  va fi plasat pe un nivel superior (un nivel cu număr strict mai mic) față de vîrful  $y$  ;
  - nivelurile sunt multimi maximale având proprietăatile de mai sus.

### Exempli

Să considerăm graful orientat din figura următoare

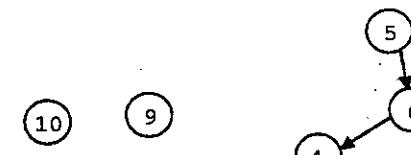


Observăm că pe nivelul 1 putem plasa numai vârfurile care au gradul interior 0. Prin urmare, nivelul 1 este format din vârfurile {2, 7, 11}. Odată ce am plasat vârfurile pe primul nivel, le eliminăm din graf împreună cu arcele incidente cu acestea și reducem problema la subgraful care se obține.

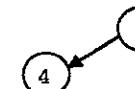


Observăm că pe nivelul al doilea putem păsa numai vârfurile care au gradul interior 0 în subgraful obținut prin eliminarea vârfurilor de pe primul nivel. Deci al doilea nivel este format din vârfurile: {3, 1, 8, 12}.

Pe nivelul al treilea putem plasa toate vârfurile care au gradul interior o în subgraful obținut prin eliminarea vârfurilor de pe primele două niveluri:



Deci pe nivelul al treilea se află vârfurile {5, 9, 10}



Nivelul 4 va fi {6}, iar ultimul nivel {4}

#### *Descrierea algoritmului*

Algoritmul aplicat pe exemplu poate fi descris succint astfel

Cât timp există vârfuri în graf, se repetă

- se plasează pe nivelul curent toate vârfurile cu gradul interior 0;
  - se elimină din graf toate vârfurile plasate pe nivelul curent, împreună cu arcele incidente cu acestea.

### Reprezentarea informațiilor

1. Reprezentăm graful prin liste de adiacență.
2. Vom utiliza un vector `np`, în care pentru fiecare vârf din graf reținem valoarea -1 dacă vârful a fost deja plasat pe un nivel sau numărul predecesorilor săi care nu au fost încă plasați pe niveluri (gradul interior al vârfului în subgraful curent).
3. Vom utiliza un vector denumit `Nivel`, în care vom reține pentru fiecare vârf nivelul pe care se află.

La fiecare pas plasăm pe nivelul curent (în vectorul `Nivel`) toate vârfurile  $i$  care nu mai au predecesori neplasați pe niveluri ( $np[i] = 0$ ). Marcăm apoi vârfurile de pe nivelul curent cu -1 în vectorul `np` și decrementăm numărul predecesorilor neplasați pe niveluri ai succesorilor lor direcți. Algoritmul se repetă până când toate vârfurile grafului sunt plasate pe niveluri.

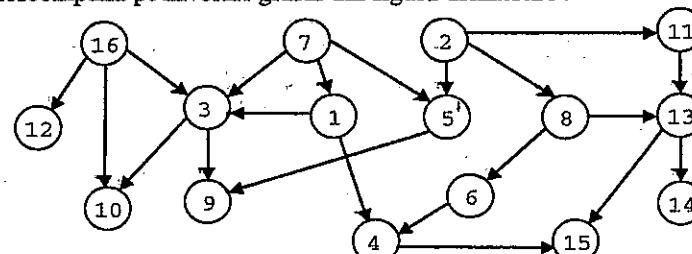
```
#include <stdio.h>
#define NMax 101
int n;
int A[NMax][NMax];
int np[NMax], Nivel[NMax];

void Citire(void);
int main()
{int i, j, total=0, nr_nivel=0;
Citire();
while (total<n)
{printf("Nivelul %d:", ++nr_nivel);
for (i=1; i<=n; i++)
if (!np[i])
{Nivel[i]=nr_nivel;
printf(" %d", i);
total++;
}
printf("\n");
for (i=1; i<=n; i++)
if (Nivel[i]==nr_nivel)
//eliminam varful i din graf
{np[i]=-1;
for (j=1; j<=A[i][0]; j++)
np[A[i][j]]--;
}
}
return 0;
}

void Citire()
{FILE * fin=fopen("graf.in", "r");
int x, y, m, i;
fscanf(fin, "%d %d", &n, &m);
for (i=0; i<m; i++)
{ffscanf(fin, "%d %d", &x, &y);
A[x][0]++;
A[x][A[x][0]]=y;
np[y]++;
}
}
```

### Exerciții propuse

1. Să se descompună pe niveluri graful din figura următoare:



2. Modificați programul astfel încât, în cazul în care graful conține circuite, să se afișeze un mesaj de eroare.

### 1.13. Sortare topologică

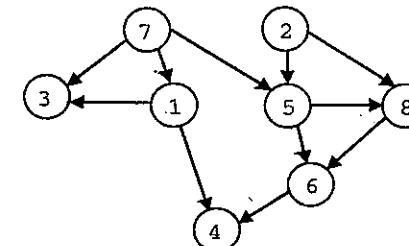
Fie  $G=(V, E)$  un graf orientat fără circuite. Să se ordoneze vârfurile grafului astfel încât, dacă există arc de la vârful  $x$  la vârful  $y$ , atunci vârful  $x$  să fie plasat în vectorul ordonat înaintea vârfului  $y$ .

#### Observații

1. Arcele din graf stabilesc o relație de ordine parțială între vârfurile grafului. Practic, această problemă solicită ordonarea elementelor unei mulțimi între care există o relație de ordine parțială.
2. Ordinea topologică a vârfurilor nu este unică.

#### Exemplu

Să considerăm graful :



Pe prima poziție în vectorul ordonat putem păsa vârful 2 sau vârful 7, deoarece sunt singurele vârfuri cu gradul interior 0. Vom alege vârful 2. Vârful 2 fiind deja plasat în vectorul ordonat, îl putem elimina din graf împreună cu toate arcele incidente cu el. Obținem un subgraf cu un vârf mai puțin și aplicăm în continuare același procedeu pentru subgraful obținut.

În final, o soluție posibilă este : {2, 7, 1, 3, 5, 8, 6, 4}.

Pe acest exemplu am aplicat un algoritm foarte asemănător cu algoritmul descompunerii pe niveluri a unui graf orientat fără circuite, cu deosebirea că, la fiecare pas, alegem un singur vârf cu gradul interior 0.

Un alt algoritm de sortare topologică a vârfurilor unui graf orientat fără circuite se bazează pe parcurgerea în adâncime a grafului.

Ca și în algoritmul Kosaraju-Sharir, se parcurge graful în adâncime și se numerotează vârfurile grafului în postordine (vârful  $x$  este numerotat după ce toți succesorii săi au fost numerotați); în vectorul postordine se memorează vârfurile în această ordine.

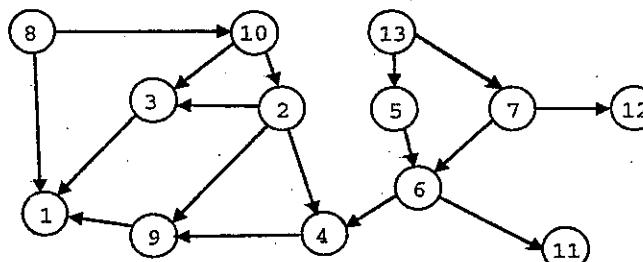
```
void DFS(int x)
{
    int i;
    viz[x]=1;
    for (i=1; i<=A[x][0]; i++)
        if (!viz[A[x][i]])
            DFS(A[x][i]);
    postordine[++nr]=x;
}
```

Evident, parcurgând vectorul postordine în sens invers (de la sfârșit către început) obținem o sortare topologică a vârfurilor grafului.

```
int main()
{
    int i;
    Citire();
    for (i=1; i<=n; i++)
        if (!viz[i])
            DFS(i);
    printf("Ordinea topologica a varfurilor: ");
    for (i=n; i>0; i--)
        printf("%d", postordine[i]);
    printf("\n");
    return 0;
}
```

### Exerciții propuse

1. Sortați topologic vârfurile grafului din figura următoare:



2. Ce condiție trebuie să îndeplinească graful pentru ca ordinea topologică a vârfurilor sale să fie unică?
3. Dați exemplu de cel puțin două situații practice, din viața cotidiană, în care să fie necesară o sortare topologică.

### 1.14. Grafuri hamiltoniene

Un graf neorientat se numește *hamiltonian* dacă el conține un ciclu hamiltonian.

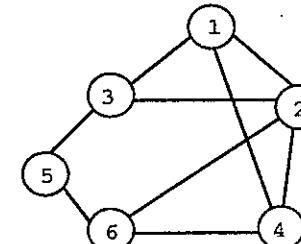
Un graf orientat se numește *hamiltonian* dacă el conține un circuit hamiltonian.

#### Observație

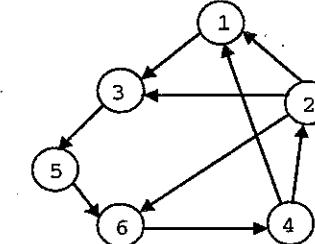
Grafurile hamiltoniene au fost studiate mai mult de 100 de ani. Ca urmare, s-au obținut o serie de rezultate remarcabile referitoare la grafurile hamiltoniene.

#### Exemple

Graf neorientat hamiltonian :



Graf orientat hamiltonian :



#### Teorema lui Dirac (1952)

Fie  $G=(V, E)$  un graf neorientat cu  $n > 2$  vârfuri. Dacă  $d(x) \geq n/2$ , pentru orice  $x$  vârf din graf, atunci graful este hamiltonian.

#### Teorema lui Ore (1960)

Fie  $G=(V, E)$  un graf neorientat cu  $n > 2$  vârfuri. Dacă suma gradelor oricărora două vârfuri neadiacente din graf este  $\geq n$ , atunci graful este hamiltonian.

Teorema Bondy-Chvátal (1972) reprezintă o generalizare a rezultatelor obținute de Øystein Ore și de Gabriel Andrew Dirac.

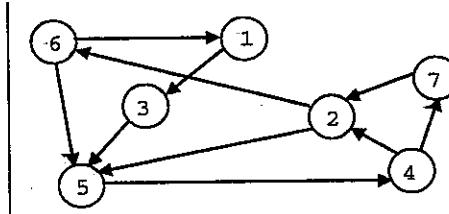
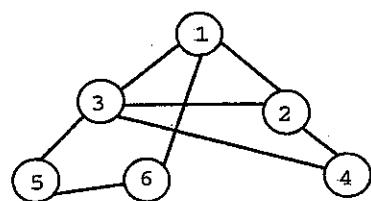
Fie  $G=(V, E)$  un graf neorientat cu  $n$  vârfuri. Se notează  $cl(G)$  și se numește închiderea grafului  $G$ , graful obținut prin adăugarea unei muchii  $[x, y]$  pentru fiecare pereche de vârfuri  $x, y$  neadiacente în graf, cu proprietatea că  $d(x)+d(y) \geq n$ .

**Teorema Bondy-Chvátal**

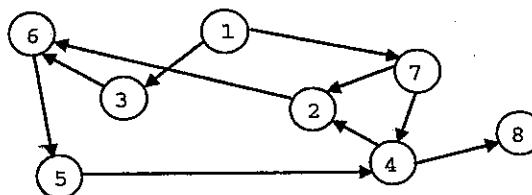
Un graf neorientat este hamiltonian dacă și numai dacă închiderea sa este graf hamiltonian.

**Exerciții propuse**

1. Să se stabilească dacă grafurile din figurile următoare sunt hamiltoniene.



2. Adăugând un număr minim de arce, transformați graful următor în graf hamiltonian:



3. Scrieți un program care să stabilească dacă un graf (orientat sau neorientat) dat este sau nu hamiltonian.  
4. Scrieți un program care să genereze toate drumurile hamiltoniene ale unui graf orientat dat.

**1.15. Grafuri euleriene**

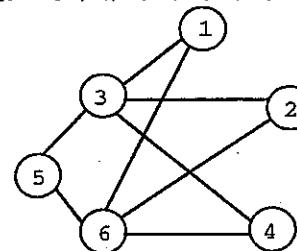
Un graf neorientat se numește *eulerian*<sup>2</sup> dacă graful conține un ciclu eulerian.

Un graf orientat se numește *eulerian* dacă graful conține un circuit eulerian.

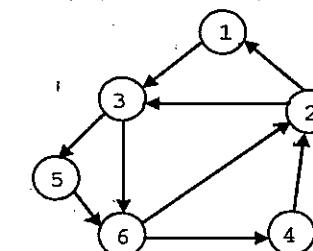
2. Denumirea acestor grafuri provine de la numele matematicianului elvețian Leonhard Euler (1707-1783). În 1736, Euler a formulat o problemă care a devenit celebră, problema podurilor din Königsberg. Orașul Königsberg din Prusia (astăzi denumit Kaliningrad, Rusia) este traversat de râul Pregel, care are două mari insule, conectate între ele și de cele două maluri prin 7 poduri. Euler a demonstrat că nu există nici un traseu care să traverseze fiecare pod o singură dată și care să revină la punctul de plecare.

**Exemple**

Graf neorientat eulerian (un ciclu eulerian este: [1, 3, 2, 6, 4, 3, 5, 6, 1]):



Graf orientat eulerian (un circuit eulerian este: (1, 3, 6, 2, 3, 5, 6, 4, 2, 1)):

**Teoremă de caracterizare a grafurilor neorientate euleriene**

Fie un graf neorientat fără vîrfuri izolate. Graful este eulerian dacă și numai dacă este conex și toate vîrfurile sale au grad par.

**Demonstrație**

Să presupunem că graful este eulerian; să demonstrează că este conex și că toate vîrfurile grafului au grad par.

Graful este eulerian ceea ce implică faptul că există un ciclu eulerian în graf. Fie  $x$  și  $y$  două vîrfuri oarecare din graf. Deoarece graful nu are vîrfuri izolate, deducem că există o muchie  $[x, a]$ , incidentă cu  $x$ , și o muchie  $[y, b]$ , incidentă cu  $y$ . Aceste muchii trebuie să aparțină ciclului eulerian, deci și vîrfurile  $x$  și  $y$  aparțin ciclului; prin urmare, există lanț de la  $x$  la  $y$ . Cum vîrfurile  $x$  și  $y$  erau vîrfuri oarecare din graf, deducem că graful este conex.

Fie  $x$  un vîrf din graf și  $a$  numărul de apariții ale lui  $x$  în ciclul eulerian. La fiecare apariție s-a utilizat o muchie pentru a „intra” în  $x$  și o muchie pentru a „ieși” din  $x$ , prin urmare numărul de muchii incidente cu  $x$  este  $2a$ , cu alte cuvinte, gradul vîrfului  $x$  este număr par.

Să considerăm acum că graful este conex și toate gradele vîrfurilor sunt pare. Să demonstrează că graful este eulerian.

Construim un ciclu în graf, plecând de la un vîrf oarecare al grafului, notat  $x_1$ ; deoarece  $x_1$  nu poate fi izolat, deducem că există o muchie incidentă cu  $x_1$ , să o notăm  $[x_1, x_2]$ . Vîrful  $x_2$  are grad par, prin urmare, dacă am utilizat muchia  $[x_1, x_2]$  pentru a „intra” în vîrful  $x_2$ , există cel puțin o muchie – să o notăm  $[x_2, x_3]$  – care ne permite să „ieșim” din vîrful  $x_2$ . Multimea muchiilor grafului fiind finită, după un număr oarecare de deplasări de-a lungul muchiilor grafului urmând acest procedeu, vom reveni în vîrful de plecare  $x_1$ , obținând astfel un ciclu.

Ciclul  $C$  astfel determinat poate să fie eulerian, caz în care problema este rezolvată. Dacă însă ciclul  $C$  nu este eulerian, deducem că multimea  $M$  a muchiilor din graf care nu aparțin acestui ciclu este nevidă. Graful dat fiind conex, deducem că există cel puțin o muchie în mulțimea  $M$  incidentă cu un vîrf de pe ciclul  $C$  (în caz

contrar, ciclul  $C$  ar forma o componentă conexă distinctă a grafului, ceea ce contrazice ipoteza.

Să considerăm o astfel de muchie din mulțimea  $M$ , notată  $[y_1, y_2]$ , vârful  $y_1$  aparținând ciclului  $C$ . Vom construi un nou ciclu, plecând din vârful  $y_1$ , deplasându-ne pe muchiile din mulțimea  $M$ , fără să trece de două ori prin aceeași muchie (acest lucru este posibil datorită parității gradelor vârfurilor). Numărul de muchii din  $M$  fiind finit, după un număr oarecare de deplasări vom reveni în vârful de plecare  $y_1$ .

Am obținut astfel un nou ciclu  $C_1$ , care are un vârf comun cu ciclul  $C$  și mulțimile muchiilor disjuncte. Reunind ciclurile  $C$  și  $C_1$  obținem un ciclu cu număr mai mare de muchii decât ciclul inițial.

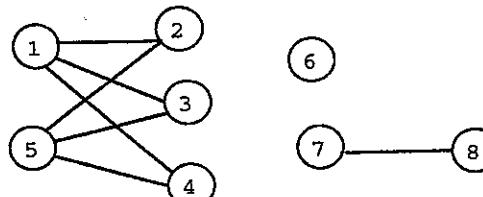
Procedeul se repetă până când numărul de muchii de pe ciclul astfel obținut este maxim (mulțimea  $M$  a muchiilor din graf care nu aparțin ciclului este vidă), deci ciclul obținut este eulerian.

### *Teoremă de caracterizare a grafurilor orientate eulériene*

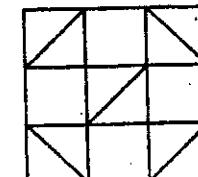
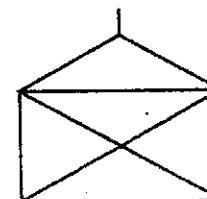
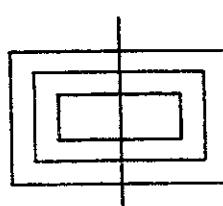
Fie un graf orientat fără vârfuri izolate. Graful este eulerian dacă și numai dacă este conex și, pentru orice vârf din graf, gradul interior este egal cu gradul exterior.

### *Exerciții propuse*

- Adăugați un număr minim de muchii în graful din figura următoare, astfel încât graful să devină eulerian :



- Care dintre figurile următoare poate fi desenată fără ridica creionul de pe hârtie (fiecare linie fiind trasată o singură dată)?



- Care dintre următoarele afirmații sunt adevărate :

- a. Un graf eulerian nu poate fi hamiltonian.
  - b. Orice graf eulerian nu trebuie să conțină vârfuri izolate.
  - c. Orice graf hamiltonian nu trebuie să conțină vârfuri izolate.
  - d. Orice graf conex admite cel puțin un arbore parțial.
  - e. Condiția necesară și suficientă ca un graf să fie hamiltonian este ca gradul fiecărui vârf să fie  $n-1$  (unde  $n$  este numărul de vârfuri din graf) ?
- Scrieți un program care să testeze dacă un graf este eulerian.
  - Utilizând algoritmul constructiv din demonstrația teoremei de caracterizare a grafurilor neorientate eulériene, scrieți un program care să determine un ciclu eulerian al unui graf dat.

### **1.16. Drumuri minime în graf**

Ca aplicație a parcurgerii în lățime am determinat cel mai scurt drum de la un anumit vârf al grafului, la celelalte vârfuri din graf, lungimea unui drum fiind măsurată în număr de arce.

În practică ne confruntăm frecvent cu situații în care arcele au asociate un cost, iar lungimea unui drum este considerată suma costurilor arcelor din care este compus drumul.

Dacă dorim să ajungem de la Iași la București, cel mai scurt drum nu este cel în care utilizăm un număr minim de șosele, ci este cel în care suma lungimilor șoselelor parcuse este minimă.

Dacă dorim să ajungem cu avionul de la Iași la Melbourne, cheltuind cât mai puțini bani, cel mai convenabil drum nu este cel în care schimbăm un număr minim de curse aeriene, ci cel în care suma costurilor biletelor este minimă.

### *Determinarea drumurilor de cost minim de la un vârf la celelalte vârfuri ale grafului*

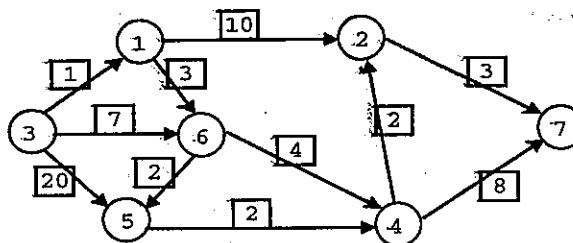
Fie  $G = (V, E)$  un graf orientat sau neorientat și  $c : E \rightarrow \mathbb{R}$ , o funcție prin care se asociază fiecărui arc/muchie din graf un număr real pozitiv denumit cost.

Să considerăm  $x_0$  un vârf al grafului, denumit vârf de start. Să se determine căte un drum de cost minim de la  $x_0$  la fiecare dintre celelalte vârfuri ale grafului.

#### *Algoritmul lui Dijkstra<sup>3</sup>*

Vom explica algoritmul aplicându-l pas cu pas pe graful ponderat din figura următoare, considerând vârful de start 3.

3. Edsger Wybe Dijkstra (1930-2002), cercetător olandez cunoscut în special pentru algoritmul care îi poartă numele, a avut contribuții remarcabile în domeniul sistemelor distribuite și al verificării formale a programelor.



Graful va fi reprezentat prin matricea costurilor c.

La momentul initial, singurul vîrf pentru care drumul de cost minim este cunoscut este vîrful 3. Vom construi o mulțime  $M$  ce reține vâfurile pentru care drumul de cost minim este deja calculat.

La fiecare moment este necesar să cunoaștem pentru fiecare vîrf  $x$  costul drumului de cost minim de la  $x_0$  la  $x$  determinat până în momentul respectiv, drum care va trece numai prin vârfuri din mulțimea  $M$ . În acest scop, vom utiliza un vector  $d$  cu  $n$  componente:  $d[x] =$  costul drumului de cost minim de la  $x_0$  la  $x$ , drum care trece doar prin vârfuri din mulțimea  $M$ .

Pentru a reconstituia drumul de cost minim vom reține pentru fiecare vîrf din graf vîrful care îl precedă pe drumul de cost minim. Pentru aceasta vom utiliza vectorul  $pre$  cu  $n$  componente:  $pre[x] =$  vîrful care îl precedă imediat pe  $x$  pe drumul minim de la  $x_0$  la  $x$ , dacă  $x \neq x_0$ ;  $pre[x_0] = 0$ . Evident, la fiecare pas  $pre[x]$  trebuie să aparțină mulțimii  $M$ , pentru orice  $x \neq x_0$ .

**Initial:**

$$M = \{3\}.$$

$$d = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 1 & \infty & 0 & \infty & 20 & 7 & \infty \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

$$pre = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 3 & 3 & 0 & 3 & 3 & 3 & 3 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

**Pasul 1:**

Selectăm un vîrf din graf care nu a mai fost selectat și pentru care distanța memorată în vectorul  $d$  de la vîrful 3 este cea mai mică. Acest vîrf este vîrful 1. Selectând vîrful 1 este posibil ca distanțele la celelalte vârfuri neselectate să se optimizeze, pentru că am putea obține un drum de cost mai mic care trece prin vîrful 1. Din acest motiv, pentru fiecare vîrf  $x$  neselectat verificăm dacă distanța  $d[x]$  este mai mare decât distanța până la vîrful 1 ( $d[1]$ ) la care se adaugă costul arcului  $(1, x)$ . În caz afirmativ,  $d[x]$  se modifică, iar vîrful care îl precedă pe  $x$  pe drumul minim de la 3 la  $x$  va fi 1.

Observăm că  $d[6]=7>d[1]+c[1][6]=1+3=4$ , deci drumul  $(3, 1, 6)$  are cost mai mic decât drumul  $(3, 6)$ . De asemenea,  $d[2]=\infty$  (deoarece nu există arcul  $(3, 2)$ ), dar trecând prin 1 obținem costul  $d[1]+c[1][2]=1+10=11$ .

Prin urmare, după pasul 1:

$$M = \{3, 1\}.$$

$$d = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 1 & 11 & 0 & \infty & 20 & 4 & \infty \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

$$pre = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 3 & 1 & 0 & 3 & 3 & 1 & 3 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

**Pasul 2:**

Selectăm un vîrf din graf care nu a mai fost selectat și pentru care distanța de la vîrful 3, memorată în vectorul  $d$ , este cea mai mică. Acest vîrf este vîrful 6. Selectând vîrful 6, unele dintre distanțele la celelalte vârfuri neselectate se vor optimiza. Observăm că  $d[5]=20>d[6]+c[6][5]=4+2=6$ , adică drumul  $(3, 1, 6, 5)$  are cost mai mic decât drumul  $(3, 5)$ . De asemenea,  $d[4]=\infty$  (deoarece nu există arcul  $(3, 4)$ ), dar trecând prin 6 obținem costul  $d[6]+c[6][4]=4+4=8$ .

Prin urmare, după pasul 2:

$$M = \{3, 1, 6\}.$$

$$d = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 1 & 11 & 0 & 8 & 6 & 4 & \infty \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

$$pre = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 3 & 1 & 0 & 6 & 6 & 1 & 3 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

**Pasul 3:**

Selectăm un vîrf din graf care nu a mai fost selectat și pentru care distanța de la vîrful 3, memorată în vectorul  $d$ , este cea mai mică. Acest vîrf este vîrful 5. Selectând vîrful 5, nici una dintre distanțele la celelalte vârfuri neselectate nu se va optimiza. Distanța până la vîrful 4, trecând prin 5, este aceeași cu distanța curentă, ceea ce înseamnă că există mai multe drumuri de cost minim de la 3 la 4.

Prin urmare, după pasul 3:

$$M = \{3, 1, 6, 5\}.$$

$$d = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 1 & 11 & 0 & 8 & 6 & 4 & \infty \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

$$pre = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 3 & 1 & 0 & 6 & 6 & 1 & 3 \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\ \hline \end{array}$$

**Pasul 4:**

Selectăm un vîrf din graf care nu a mai fost selectat și pentru care distanța de la vîrful 3, memorată în vectorul  $d$ , este cea mai mică. Acest vîrf este vîrful 4. Selectând vîrful 4, se vor optimiza distanțele până la vârfurile 2 și 7:

$$d[2]=11>d[4]+c[4][2]=8+2=10.$$

$$d[7]=\infty>d[4]+c[4][7]=8+8=16.$$

Prin urmare, după pasul 4 :

$$M = \{3, 1, 6, 5, 4\}.$$

d=	1	10	0	8	6	4	16
	1	2	3	4	5	6	7

pre=	3	4	0	6	6	1	4
	1	2	3	4	5	6	7

Pasul 5 :

Selectăm un vârf din graf care nu a mai fost selectat și pentru care distanța de la vârful 3, memorată în vectorul d, este cea mai mică. Acest vârf este vârful 2. Selectând vârful 2 se va optimiza distanța până la vârful 7 :

$$d[7] = 16 > d[2] + C[2][7] = 10 + 3 = 13.$$

Prin urmare, după pasul 5 :

$$M = \{3, 1, 6, 5, 4, 2\}.$$

d=	1	10	0	8	6	4	13
	1	2	3	4	5	6	7

pre=	3	4	0	6	6	1	2
	1	2	3	4	5	6	7

Pasul 6 :

Se selectează ultimul vârf neselectat, vârful 7. Costurile minime sunt memorate în vectorul d, iar drumurile de cost minim pot fi reconstituite utilizând informațiile din vectorul pre. De exemplu, pentru a reconstituîi drumul de cost minim de la 3 la 7 vom proceda în sens invers, începând cu 7. Predecesorul lui 7 este 2, predecesorul lui 2 este 4, predecesorul lui 4 este 6, predecesorul lui 6 este 1, iar al lui 1 este 3. Am obținut secvența 7, 2, 4, 6, 1, 3, care reprezintă un drum de cost minim de la 3 la 7 scris invers.

Descrierea algoritmului

Initializare :

Se citesc n, C, x0

M = {x0}

d[x] = C[x0][x], pentru orice x = 1, 2, ..., n

pre[x] = x0, pentru x = 1, 2, ..., n, x ≠ x0;

pre[x0] = 0;

Se repetă de n-1 ori :

Se determină x un vârf din graf neselectat (x ∈ M), astfel încât :

d[x] = min(d[y] | y ∈ M)

M = M ∪ {x}

Se actualizează distanțele către celelalte vârfuri neselectate din graf, y ∈ M :

dacă d[y] > d[x] + C[x][y] atunci

d[y] = d[x] + C[x][y]

pre[y] = x

Implementarea algoritmului

```
#include <iostream.h>
#define NMaxVf 50
#define Inf 10000

int n, x0;
double C[NMaxVf][NMaxVf];
int pre[NMaxVf], M[NMaxVf];
double d[NMaxVf];

void Initializare();
void Afisare();

int main()
{
    int i, VfMin, j;
    double dMin;
    Initializare();
    for (j=1; j<n; j++)
        {dMin=Inf;
         for (i=1; i<=n; i++)
             if (!M[i] && dMin>d[i])
                 {dMin=d[i];
                  VfMin=i; }
        M[VfMin]=1;
        for (i=1; i<=n; i++)
            if (!M[i] && d[i]>dMin+C[VfMin][i])
                {pre[i]=VfMin;
                 d[i]=dMin+C[VfMin][i]; }
        }
    Afisare();
    return 0;
}

void Initializare()
{
    int i, j, m, x, y;
    double c;
    ifstream fin("graf.in");
    fin>>n>>m>>x0;
    for (i=1; i<=n; i++)
        for (j=i+1; j<=n; j++)
            C[j][i]=C[i][j]=Inf;
    for (i=1; i<=m; i++)
        {fin>>x>>y>>c;
         C[x][y]=c; }
    for (i=1; i<=n; i++)
        {d[i]=C[x0][i]; pre[i]=x0; }
    M[x0]=1; pre[x0]=0;
    fin.close();
}
```

```

void Afisare()
{
    int i, j, lg, dr[NMaxVf];
    for (i=1; i<=n; i++)
        if (i!=x0)
            cout<<"Costul drumului de cost minim de la "<<x0<<
                " la "<<i<<" este "<<d[i]<<endl;
    cout<<"Drumul de cost minim:";
    dr[0]=i; lg=0;
    while (pre[dr[lg]])
        {
            lg++;
            dr[lg]=pre[dr[lg-1]];
        }
    for (j=lg; j>=0; j--) cout<<' '<<dr[j];
    cout<<endl;
}

```

#### Observații

- Complexitatea algoritmului lui Dijkstra este  $O(n^2)$ .
- Algoritmul lui Dijkstra funcționează și în cazul în care costurile arcelor sunt numere negative, dar în graf nu există circuite de cost total negativ. În cazul în care există circuite de cost negativ, costul minim pentru orice vârf care aparține circuitului ar putea fi considerat  $-\infty$ , dar algoritmul lui Dijkstra nu detectează circuitele de cost negativ.

#### Algoritmul Bellman-Ford

Algoritmul Bellman-Ford determină drumurile de cost minim de la un vârf din graf la fiecare dintre celelalte vârfuri, identificând eventualele circuite de cost negativ.

#### Descrierea algoritmului

Se vor utiliza aceleasi structuri de date ca la algoritmul lui Dijkstra:

- Graful este reprezentat prin matricea costurilor  $C$ .
- Costurile drumurilor de cost minim determinate până la un moment dat sunt reținute în vectorul  $d$ .
- Pentru a reconstitui drumul de cost minim se utilizează vectorul  $pre$ , în care reținem pentru fiecare vârf din graf predecesorul său pe drumul de cost minim de la vârful de start.

Algoritmul are două etape: în prima etapă se parcurg toate arcele grafului și pentru fiecare arc din graf se verifică dacă optimizează costul drumului de cost minim de la vârful de start la extremitatea sa finală; această verificare se repetă de  $n$  ori. În cea de-a două etapă se verifică dacă există circuite de cost negativ.

Se repetă de  $n$  ori:

Pentru fiecare arc  $(x, y)$  din graf:  
 Dacă  $d[y] > d[x] + C[x][y]$  atunci  
 $d[y] = d[x] + C[x][y];$   
 $pre[y] = x;$

Pentru fiecare arc  $(x, y)$  din graf

Dacă  $d[y] > d[x] + C[x][y]$  atunci  
 Scrie „Există circuite de cost negativ”. Stop

#### Implementarea algoritmului

Deoarece funcțiile de inițializare și de afișare sunt aceleasi ca în implementarea algoritmului lui Dijkstra, vom prezenta numai funcția Bellman\_Ford(). Această funcție returnează valoarea 0 dacă graful conține un circuit de cost negativ și respectiv 1 în caz contrar.

```

int Bellman_Ford()
{
    int i, j, k;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            for (k=1; k<=n; k++)
                if (C[j][k] != Inf && d[k] > d[j] + C[j][k])
                    d[k] = d[j] + C[j][k];
                    pre[k] = j;

    for (j=1; j<=n; j++)
        for (k=1; k<=n; k++)
            if (C[j][k] != Inf && d[k] > d[j] + C[j][k])
                return 0;
    return 1;
}

```

#### Observații

Complexitatea algoritmului Bellman-Ford este  $O(n^3)$ .

#### Determinarea costurilor drumurilor de cost minim între oricare două vârfuri ale unui graf

Fie  $G=(V, E)$  un graf orientat sau neorientat și  $c: E \rightarrow \mathbb{R}$ , o funcție prin care se asociază fiecărui arc/muchie din graf un număr real pozitiv denumit cost.

Pentru fiecare pereche de vârfuri din graf  $x, y$  să se determine costul drumului de cost minim de la  $x$  la  $y$ .

#### Algoritmul Floyd-Warshall

Algoritmul Floyd-Warshall se bazează pe o idee similară cu ideea algoritmului Roy-Warshall: considerăm toate vârfurile intermedii  $z$  ( $z=1, 2, \dots, n$ ) și verificăm pentru fiecare pereche de vârfuri din graf  $x, y$  dacă trecând prin  $z$  se optimizează costul de la  $x$  la  $y$ ; în caz afirmativ, costul drumului de la  $x$  la  $y$  se modifică. Observăm că după pasul  $z$  în matricea costurilor, pe poziția  $(x, y)$ , se află costul drumului de cost minim de la  $x$  la  $y$  care trece numai prin vârfuri intermedii din mulțimea  $\{1, 2, \dots, z\}$ .

#### Descrierea algoritmului

Vom reprezenta graful prin matricea costurilor  $C$  și vom obține costurile minime în aceeași matrice.

```

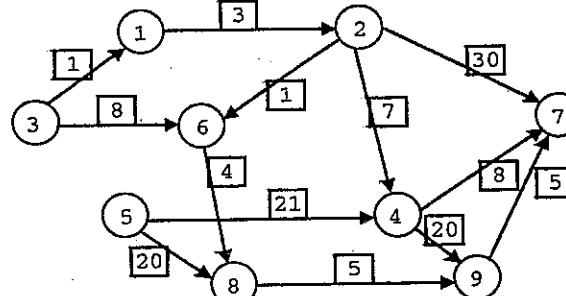
for (z=1; z<=n; z++)
    for (x=1; x<=n; x++)
        for (y=1; y<=n; y++)
            d[x][y]=min(d[x][z]+d[z][y], d[x][y])
    
```

**Observații**

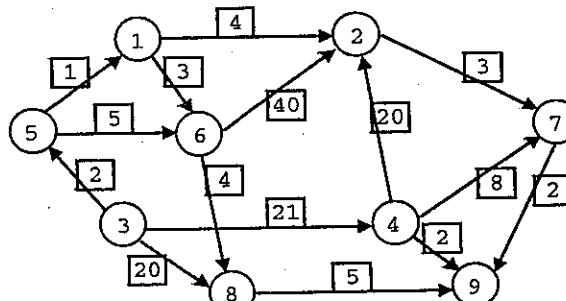
1. Complexitatea algoritmului este  $O(n^3)$ .
2. Algoritmul are la bază metoda programării dinamice.

**Exerciții propuse**

1. Se consideră graful din figura următoare. Să se indice două drumeuri de cost minim de la vârful 3 la vârful 7.



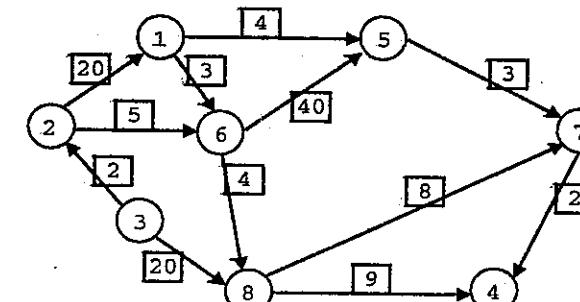
2. Se consideră graful ponderat din figura următoare și vârful de start 5. Determinați câte un drum de cost minim de la vârful 5 la fiecare vârf al grafului, aplicând algoritmul lui Dijkstra.



3. După aplicarea algoritmului lui Dijkstra, vectorul pre are următorul conținut. Să se reconstituie drumeul de la vârful de plecare la vârful 5 indicând în ordine vârfurile parcurse.

pre =	8	1	4	1	7	2	6	0	4	2
	1	2	3	4	5	6	7	8	9	10

4. Care va fi al treilea vârf selectat în algoritmul lui Dijkstra pentru următorul graf și vârful de start 2?



5. Se consideră graful de la exercițiul 4 și vârful de start 2. Care va fi conținutul vectorului d după prima parcursare a tuturor arcelor grafului și executarea tuturor optimizărilor necesare în algoritmul Bellman-Ford?
6. Se consideră graful de la exercițiul 4. Care va fi conținutul matricei costurilor după executarea primilor doi pași (pentru  $z=1$  și  $z=2$ ) din algoritmul Floyd-Warshall?
7. Modificați algoritmul lui Dijkstra astfel încât să determine toate drumeurile de cost minim de la vârful de start la fiecare vârf din graf.
8. Modificați algoritmul Floyd-Warshall astfel încât să genereze și un drum de cost minim pentru fiecare pereche de vârfuri din graf.

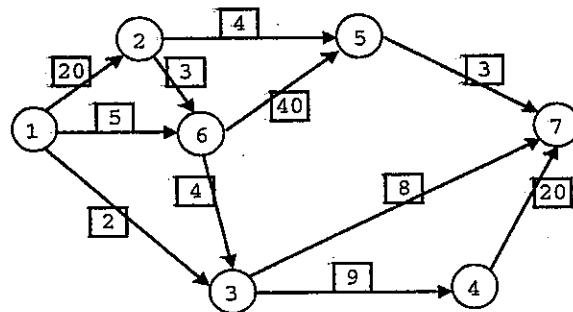
**1.17. Rețele de transport**

Se numește *rețea de transport* un graf orientat  $G=(V, E)$  care satisfac următoarele condiții:

1. există un singur vârf în graf cu gradul interior 0 (acest vârf este denumit vârf-sursă sau intrare în rețea);
2. există un singur vârf în graf cu gradul exterior 0 (acest vârf este denumit vârf-destinație sau ieșire din rețea);
3. graful este conex și există drumeuri de la vârful-sursă la vârful-destinație;
4. există o funcție  $c: E \rightarrow \mathbb{R}$ , prin care se asociază fiecărui arc din graf un număr real pozitiv denumit capacitatea arcului respectiv.

**Exemplu**

Graful orientat din figura următoare reprezintă o rețea de transport (vârful 1 este intrarea rețelei, iar vârful 7 este ieșirea rețelei).



Se numește *flux* în rețea de transport o funcție  $f: E \rightarrow \mathbb{R}_+$  care îndeplinește următoarele condiții:

*Condiția de mărginire a fluxului*:  $f(x, y) \leq c(x, y)$  pentru orice  $(x, y) \in E$  (pentru orice arc din graf valoarea fluxului nu poate depăși capacitatea arcului respectiv).

*Condiția de conservare a fluxului*: pentru orice vârf  $x$  din graf, exceptând intrarea și ieșirea rețelei, suma fluxurilor arcelor care au extremitatea inițială vârful  $x$  este egală cu suma fluxurilor arcelor care au ca extremitate finală vârful  $x$ :

$$\sum_{(x,y) \in E} f(x, y) = \sum_{(y,x) \in E} f(y, x)$$

**Observație**

Datorită condiției de conservare a fluxului în rețea, suma fluxurilor arcelor care au ca extremitate inițială vârful s este egală cu suma fluxurilor arcelor care au ca extremitate finală vârful d (această sumă este denumită valoarea fluxului).

**Problema**

Fie  $G=(V, E, c)$  o rețea de transport. Să se determine un flux de valoare maximă.

**Algoritmul Ford-Fulkerson**

Inițializăm fluxul în rețea cu 0.

Cât timp este posibil:

- determinăm un lanț de ameliorare a fluxului,
- ameliorăm fluxul de-a lungul acestui drum.

Pentru a determina un lanț de ameliorare, Edmonds și Karp au implementat următorul procedeu de marcat, bazat pe o parcurgere în lățime a grafului, începând cu intrarea rețelei.

Se marchează intrarea rețelei cu +.

Fie  $x$  un vârf nemarcat. Se marchează cu  $+x$  orice vârf  $y$  nemarcat cu proprietatea că există un arc nesaturat de la  $x$  la  $y$  ( $(x, y) \in E$  și  $f(x, y) < c(x, y)$ ).

Se marchează cu  $-x$  orice vârf nemarcat  $y$  cu proprietatea că există un arc cu flux nul de la  $y$  la  $x$  ( $(y, x) \in E$  și  $f(y, x) > 0$ ). Dacă prin acest procedeu de marcat s-a marcat ieșirea rețelei, atunci fluxul curent nu este maxim.

Pe baza etichetelor vârfurilor reconstituim un lanț de la intrarea rețelei, până la ieșire și modificăm fluxurile arcelor care constituie acest lanț după cum urmează.

Să notăm:

$$a = \min \{c(x, y) - f(x, y) \mid y \text{ marcat cu } +x, (x, y) \in L\}$$

$$b = \min \{f(y, x) \mid y \text{ marcat } -x, (y, x) \in L\}$$

$$v = \min \{a, b\}. \text{ Din modul de determinare a lanțului, } v > 0.$$

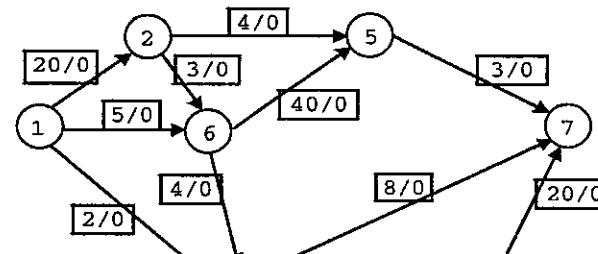
Vom mări cu  $v$  fluxul arcelor care aparțin lanțului și sunt orientate în sensul sursă → destinație (adică arcele  $(x, y)$  unde  $y$  este marcat  $+x$ ).

Pentru a respecta condiția de conservare a fluxului, vom micșora cu  $v$  fluxul arcelor care aparțin lanțului și sunt orientate în sensul destinație → sursă (adică arcele  $(y, x)$  unde  $y$  este marcat  $-x$ ).

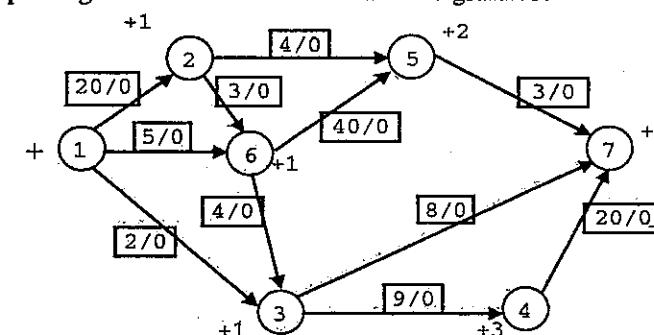
Din modul de definire a noului flux, observăm că respectă condiția de mărginire și de conservare și are valoarea cu  $v$  mai mare decât fluxul inițial.

**Exemplu**

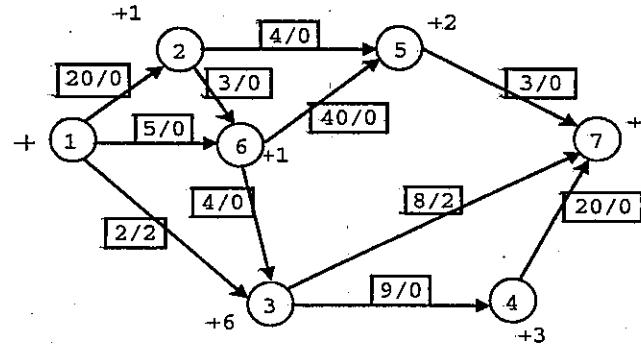
Vom aplica algoritmul pentru rețea de transport următoare. Pe fiecare arc este specificată capacitatea, urmată de fluxul arcului respectiv.



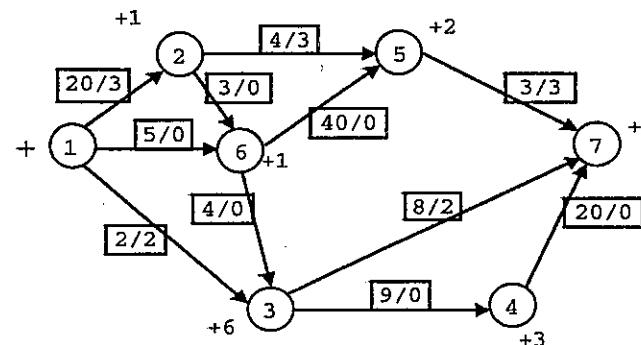
Printr-o parcurgere BFS din 1 marcăm vârfurile grafului:



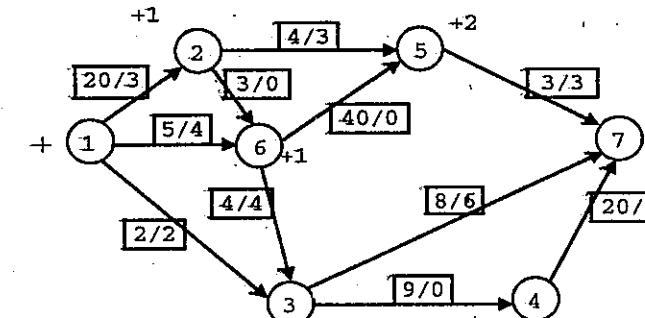
Observăm că a fost marcată ieșirea rețelei (vârful 7), reconstituim drumul de ameliorare: (1, 3, 7). Toate arcele sunt orientate în sensul sursă → destinație, calculăm  $\min\{c(1,3)-f(1,3), c(3,7)-f(3,7)\}=\min\{2, 8\}=2$ . Mărim fluxul fiecarui arc de pe acest drum cu 2 și reluăm procedeul de marcat. Arcul (1, 3) a devenit saturat (fluxul este egal cu capacitatea).



Din nou a fost marcată ieșirea rețelei, reconstituim drumul de ameliorare: (1, 2, 5, 7). Toate arcele sunt orientate în sensul sursă → destinație, calculăm  $\min\{c(1,2)-f(1,2), c(2,5)-f(2,5), c(5,7)-f(5,7)\}=\min\{20, 4, 3\}=3$ . Mărim fluxul fiecarui arc de pe acest drum cu 3 și reluăm procedeul de marcat. Arcul (5, 7) a devenit saturat.



Din nou a fost marcată ieșirea rețelei, reconstituim drumul de ameliorare: (1, 6, 3, 7). Toate arcele sunt orientate în sensul sursă → destinație, calculăm  $\min\{c(1,6)-f(1,6), c(6,3)-f(6,3), c(3,7)-f(3,7)\}=\min\{5, 4, 6\}=4$ . Mărim fluxul fiecarui arc de pe acest drum cu 4 și reluăm procedeul de marcat. Arcul (6, 3) a devenit saturat.



La această nouă parcurgere nu am reușit să marcăm vârful 7, deci 9 este fluxul maxim în rețea.

### Implementarea algoritmului

Pentru a reprezenta rețeaua de transport vom utiliza o matrice C cu n linii și n coloane ( $C[x][y]$  = capacitatea arcului  $(x, y)$ ).

Pentru a reține fluxul în rețea vom utiliza o altă matrice F cu n linii și n coloane ( $F[x][y]$  = fluxul arcului  $(x, y)$ ).

```
#include <stdio.h>
#define MAXV 50
#define min(x,y) ((x)<(y)?(x):(y))
#define abs(x) ((x)>0?x:-x)

int n, m, s, d, viz[MAXV], Q[MAXV];
int C[MAXV][MAXV]; //capacitatea fiecarui arc
int F[MAXV][MAXV]; //fluxul fiecarui arc

void citire(void);
void ek(void);
void afisare(void);
int bfs(void);

int main()
{
    citire();
    ek();
    afisare();
    return 0;
}

void citire()
(FILE * fin=fopen("graf.in", "r"));
int i, x, y, c;
fscanf(fin,"%d %d %d %d", &n, &m, &s, &d);
for (i=0; i<m; i++)
    fscanf(fin,"%d %d %d", &x, &y, &c);
    C[x][y]=c;
}
```

```

void afisare()
{
    int i, j, vf=0;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            if (F[i][j])
                printf("Fluxul arcului %d-%d=%d\n", i, j, F[i][j]);
    for (i=1; i<=n; i++) vf+= F[i][d];
    printf ("Fluxul maxim %d\n", vf);
}

void ek()
{
    int i, a, b, lg, v;
    int L[MAXV];
    do
    {
        //marcam varfurile intr-o parcurgere in latime
        for (i=1; i<=n; i++) viz[i]=0;
        if (bfs()) return;
        //determinam lantul de ameliorare in vectorul L
        L[0]=d; lg=0;
        a=b=10000;
        while (L[lg]!=s)
        {
            ++lg;
            L[lg]=abs(viz[L[lg-1]]);
            if (viz[L[lg-1]]>0)
                a=min(a, C[L[lg]][L[lg-1]]-F[L[lg]][L[lg-1]]);
            else
                if (viz[L[lg-1]]<0)
                    b=min(b, F[L[lg-1]][L[lg]]);
        }
        v=min(a,b);
        //marim fluxul de-a lungul lantului
        for (i=lg; i>0; i--)
            if (viz[L[i-1]]>0)
                F[L[i]][L[i-1]]+=v;
            else
                F[L[i-1]][L[i]]-=v;
        }
    while (1);
}

int bfs()
//returneaza 1 daca iesirea retelei nu a fost marcata
{
    int p, u, i, x;
    Q[0]=s; p=u=0; viz[s]=1;
    while (p<=u && !viz[d])
    {
        x=Q[p++];
        for (i=1; i<=n; i++)
            if (!viz[i])
                if (F[x][i]<C[x][i])
                    (viz[i]=x, Q[++u]=i);
    }
}

```

```

        else
            if (F[i][x]>0)
                (viz[i]=-x, Q[++u]=i);
    }
    return !viz[d];
}

```

#### Observație

Complexitatea algoritmului în implementarea oferită de Edmonds-Karp este  $O(n \cdot m^2)$ .

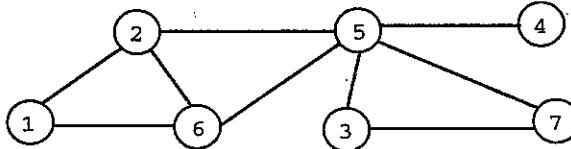
### 1.18. Cuplaj maximal în graf bipartit

Fie  $G$  un graf neorientat. Se numește *cuplaj* în graful  $G$  o mulțime de muchii cu proprietatea că fiecare vârf este incident cu cel mult o muchie din mulțime.

Cuplajul se numește *maximal* dacă numărul de muchii din mulțime este maxim.

#### Exemplu

Pentru graful următor, mulțimea  $\{[2, 6], [5, 3]\}$  este un cuplaj. Acest cuplaj nu este maximal (un cuplaj maximal ar fi:  $\{[1, 6], [5, 4], [3, 7]\}$ ).



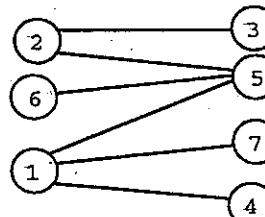
Pentru cazul în care graful este bipartit (adică mulțimea vârfurilor poate fi partionată în două mulțimi  $A, B$ , astfel încât orice muchie din graf are o extremitate în mulțimea  $A$  și cealaltă extremitate în mulțimea  $B$ ), problema poate fi redusă la determinarea fluxului maxim într-o rețea de transport.

Construim o rețea de transport în care :

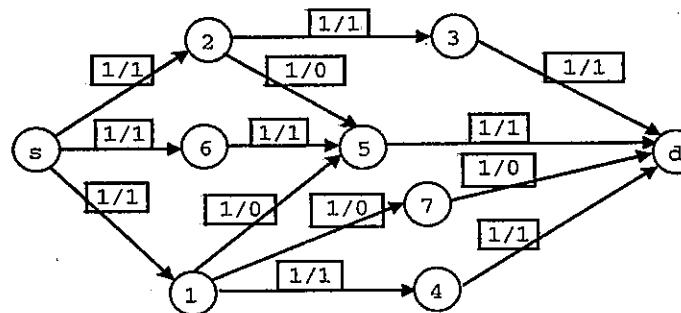
1. Mulțimea vârfurilor este mulțimea vârfurilor grafului  $G$ , la care adăugăm două vârfuri fictive  $s$  și  $d$  (sursa și destinația).
2. Mulțimea arcelor grafului este formată din :
  - arcele care au sursa drept extremitate inițială și vârfurile din mulțimea  $A$  ca extremitate finală;
  - arcele care au ca extremitate finală destinația și ca extremitate inițială fiecare vârf din mulțimea  $B$ ;
  - arcele care corespund muchiilor grafului  $G$ , considerând ca extremitate inițială vârful din mulțimea  $A$  și ca extremitate finală vârful din mulțimea  $B$ .
3. Fiecare arc va avea capacitatea 1.

**Exemplu**

Să considerăm următorul graf bipartit:



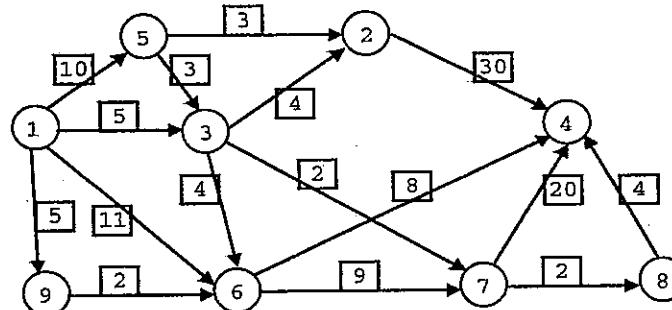
Rețeaua de transport asociată acestui graf bipartit este ilustrată în figura următoare. Am figurat și un flux maximă în rețea.



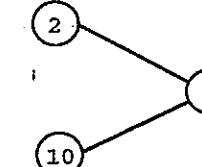
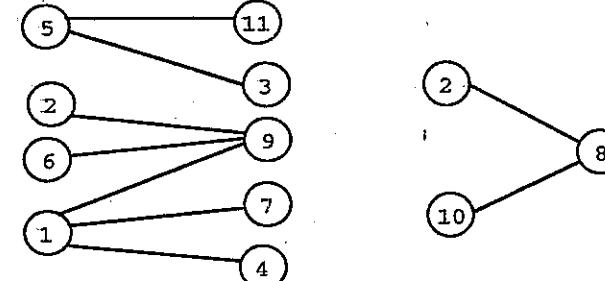
Determinând un flux maxim în această rețea de transport, determinăm practic un cuplaj în graf, format din toate muchiile corespunzătoare arcelor de la mulțimea A la mulțimea B având flux nenu.

**Exerciții propuse**

1. Determinați un flux maximal în rețeaua de transport din figura următoare aplicând algoritmul Ford-Fulkerson.



2. Determinați un cuplaj maximal în graful din figura următoare.

**1.19. Aplicații rezolvate****Bile**

Aveam N bile (numerotate de la 1 la N), oricare două bile având greutăți diferite. Pentru a descoperi bila cu greutatea mediană (deci cea de-a  $(N+1)/2$ -a bilă, în ordinea greutăților) putem utiliza o balanță cu două platane. Putem plasa căte o bilă pe fiecare platan și astfel putem afla care dintre cele două bile este mai grea. Ca urmare a rezultatelor obținute în urma unui set de căntări, putem elimina unele dintre bile, despre care putem afirma cu siguranță că nu au greutatea mediană.

De exemplu, să considerăm  $N=5$  bile, între care s-au efectuat  $M=4$  căntări, obținându-se următoarele rezultate:

- Bila 2 este mai grea decât bila 1.
- Bila 4 este mai grea decât bila 3.
- Bila 5 este mai grea decât bila 1.
- Bila 4 este mai grea decât bila 2.

Din rezultatelor de mai sus, deși nu putem determina exact bila cu greutatea mediană, putem afirma că bila 1 și bila 4 nu pot avea greutatea mediană, deoarece bilele 2, 4, 5 sunt mai grele decât bila 1, iar bilele 1, 2, 3 sunt mai ușoare decât bila 4. Prin urmare, putem elimina aceste două bile.

**Cerință**

Scrieți un program care să determine pe baza unui set de căntări dat numărul de bile care nu pot avea greutatea mediană.

**Date de intrare**

Fisierul de intrare `bile.in` conține pe prima linie numărul natural  $N$ , reprezentând numărul de bile, și numărul natural  $M$ , reprezentând numărul de căntări, separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii conține două numere naturale cuprinse între 1 și  $N$  cu semnificația „bila corespunzătoare primului număr este mai grea decât bila corespunzătoare celui de-al doilea număr”.

**Date de ieșire**

Fisierul de ieșire `bile.out` va conține o singură linie pe care se află numărul de bile care nu pot avea greutatea mediană.

**Restrictii**

N număr natural impar,  $0 < N \leq 300$ ;  
 $M \leq 5000$

**Exemple**

bile.in	bile.out
5 4	
2 1	2
4 3	
5 1	
4 2	

(campion, 2004)

**Soluție**

Asociem problemei un graf orientat astfel:

- vârfurile grafului sunt cele n bile;
- dacă bila i este mai grea decât bila j, atunci există arc în graf de la i la j.

Reprezentăm graful asociat problemei prin liste de adiacență:  $g[i]$  conține lista vârfurilor j cu proprietatea că există arc de la i la j, iar  $d[i]$  este numărul de vârfuri din lista  $g[i]$ .

Pentru a determina numărul de bile mai grele decât bila i vom parurge graful din vârful i. Dacă numărul de bile mai grele decât bila i este mai mare decât  $(n+1)/2$ , atunci bila i poate fi eliminată. În caz contrar, trebuie să determinăm numărul de bile mai ușoare decât bila i. Pentru a determina numărul de bile mai ușoare decât bila i vom construi graful transpus. Parcugând graful transpus din vârful i putem determina numărul de bile mai ușoare decât bila i. Dacă numărul de bile mai ușoare decât bila i este mai mare decât  $(n+1)/2$ , atunci bila i poate fi eliminată.

```
#include <fstream.h>
#define maxn 301
#define infile "bile.in"
#define outfile "bile.out"

int n, m, res, nrviz;
int g[maxn][maxn], gt[maxn][maxn];
int d[maxn], dt[maxn], viz[maxn];

void dfs(int);
void dfsT(int);
void citire();
void rezolvare();
void afisare();

int main()
{
    citire();
    rezolvare();
    afisare();
    return 0;
}
```

```
void citire()
{
    int tmp1, tmp2, i, j;
    for (i=0; i<maxn; i++)
        {d[i]=0; dt[i]=0;
         for (j=0; j<maxn; j++)
             g[i][j]=gt[i][j]=0; }
    ifstream indat(infile);
    indat>>n>>m;
    /* construim graful G si graful transpus GT */
    for (i=0; i<m; i++)
        {indat>>tmp1>>tmp2;
         tmp1--; tmp2--;
         g[tmp1][d[tmp1]]=tmp2; d[tmp1]++;
         gt[tmp2][dt[tmp2]]=tmp1; dt[tmp2]++;
        }
    indat.close();
}

void rezolvare()
{
    int i, j;
    for (i=0; i<n; i++)
        { /* pentru fiecare bila i */
            nrviz=0;
            for (j=0; j<n; j++) viz[j]=0;
            /* determin numarul de bile mai grele decat bila i*/
            dfs(i);
            if (nrviz>=(n+1)/2) res++;
            else
                {nrviz=0;
                 for (j=0; j<n; j++) viz[j]=0;
                 /*determin nr. de bile mai usoare decat bila i*/
                 dfsT(i);
                 if (nrviz>=(n+1)/2) res++;
                }
        }
}

void afisare()
{
    ofstream outdat(outfile);
    outdat<<res<<'\n';
    outdat.close();
}

void dfs(int i)
{
    int j;
    for (j=0; j<d[i]; j++)
        if (viz[g[i][j]]==0)
            {viz[g[i][j]]=1;
             nrviz++;
             dfs(g[i][j]); }
}
```

```

void dfsT(int i)
{
    int j;
    for (j=0; j<dt[i]; j++)
        if (viz[gt[i][j]]==0)
            {viz[gt[i][j]]=1;
             nrviz++;
             dfsT(gt[i][j]);
            }
}

```

### Cinste

Directorul școlii dorește să afle despre fiecare elev din școala dacă este cinstit (nu minte niciodată) sau este mincinos. El știe cu siguranță că măcar jumătate dintre elevi sunt cinstiți, dar ar dori să afle cu exactitate caracterul fiecărui elev.

În acest scop, numerotează elevii de la 1 la n, așază elevii în rând, într-o ordine oarecare, apoi solicită fiecare elev să își exprime părerea despre colegii situați în rând după el. Un elev cinstit va spune întotdeauna adevărul: întrebăt care e părerea lui despre un mincinos, va răspunde neîndoilenic că acesta minte, iar despre un elev cinstit, că nu minte niciodată. În schimb, un mincinos va minti întotdeauna: va spune despre un elev cinstit că minte, iar despre unul asemenea lui, că este cinstit.

Cum școala este foarte mare, este posibil ca un elev să nu își cunoască toți colegii, dar din afirmațiile elevilor se poate deduce în mod unic ordinea în care aceștia au fost așezăți în rând.

### Cerință

Scriți un program care să determine pentru fiecare elev dacă este sau nu cinstit.

### Date de intrare

Fișierul de intrare `cinste.in` conține pe prima linie două numere naturale separate printr-un spațiu  $n \ m$ , reprezentând numărul de elevi, respectiv numărul de păreri exprimate. Pe fiecare dintre următoarele  $m$  linii sunt scrise triplete de forma:  $x \ c \ y$ , unde  $x$  și  $y$  sunt numere întregi, ( $1 \leq x, y \leq n$ ),  $x$  reprezintă numărul de ordine al elevului care este interogat referitor la elevul cu numărul de ordine  $y$ ;  $c$  este un caracter și poate fi '`c`', semnificând faptul că  $x$  spune despre  $y$  că acesta este cinstit, respectiv '`m`', dacă  $x$  spune despre  $y$  că acesta este mincinos.

### Date de ieșire

Fișierul `cinste.out` va conține  $n$  linii, câte una pentru fiecare elev. Pe linia  $i$  se va scrie 1 dacă elevul  $i$  este cinstit, respectiv 0 dacă elevul  $i$  este mincinos.

### Restricții și precizări

$1 < n \leq 1000$ ;  
 $0 < m \leq 200000$

### Exemplu

cinste.in	cinste.out
5 9	1
1 c 2	1
2 c 3	1
1 m 4	0
2 m 4	0
5 c 4	
3 m 4	
1 m 5	
2 m 5	
5 m 3	

(.campion, 2005)

### Soluție

Vom asocia problemei un graf orientat astfel:

- elevii sunt vârfurile grafului;
- dacă elevul  $x$  își exprimă o opinie referitoare la elevul  $y$ , atunci există un arc în graf de la  $x$  la  $y$  (arcul este etichetat 1 dacă  $x$  spune că  $y$  este cinstit și -1 în caz contrar).

Vom reprezenta graful prin liste de adiacență alocate static. Linia  $i$  reprezintă lista de adiacență a vârfului  $i$ . În lista de adiacență memorăm o structură cu două câmpuri ( $y$  și eticheta  $1/-1$ ).

În tabloul s memorăm starea nodurilor:

$s[i] < 0 \Rightarrow$  nodul  $i$  corespunde unui elev mincinos

$s[i] > 0 \Rightarrow$  nodul  $i$  corespunde unui elev cinstit

$s[i] = 0 \Rightarrow$  nodul  $i$  corespunde unui elev a căruia stare este necunoscută.

Mai întâi sortăm topologic nodurile grafului, pentru a determina ordinea în care au fost așezăți elevii. Ordinea o vom obține în vectorul  $o$ .

Pentru început considerăm că  $o[1]$  este un elev cinstit. Parcurgem graful începând din nodul  $o[1]$ , în ordine topologică, și determinăm caracterul fiecărui elev. În final, dacă numărul de elevi cinstiți este mai mic decât numărul de elevi minciinoși, îi vom interschimba (schimbăm semnul elementelor vectorului  $s$ ).

Datorită faptului că ordinea poate fi determinată în mod unic, deducem că graful este conex.

```

#include <stdio.h>
#include <stdlib.h>
#define NMax 1001
#define MMax 8001
#define InFile "cinste.in"
#define OutFile "cinste.out"
#define sgn(x) ((x)>0?1:0);
typedef struct (int y, c;) Opinion;
int n, m, total;
Opinion * G[NMax], * GT[NMax];
int d[NMax], di[NMax], o[NMax];
int s[NMax];

```

```

void read_it(void);
void solve(void);
void write_it(void);
int dfs(int, int);
void top_sort(void);

int main()
{read_it();
 top_sort();
 solve();
 write_it();
 return 0;
}

void read_it(void)
{int i, x, y; char c;
 FILE * fin=fopen(InFile, "rt");
 fscanf(fin, "%d %d", &n, &m);
 for (i=0; i<m; i++)
 {fscanf(fin, "%d %c %d", &x, &c, &y);
 d[x]++;
 di[y]++;
 G[x]=realloc(G[x], d[x]*sizeof(Opinion));
 G[x][d[x]-1].y=y;
 GT[y]=realloc(GT[y], di[y]*sizeof(Opinion));
 GT[y][di[y]-1].y=x;
 if (c=='c') G[x][d[x]-1].c=GT[y][di[y]-1].c=1;
 else G[x][d[x]-1].c=GT[y][di[y]-1].c=-1;
 }
 fclose(fin);
}

void solve(void)
{int i, j, how;
 s[o[1]]=1;
 for (i=2; i<=n; i++)
 {how=0;
 for (j=0; j<di[o[i]]; j++)
 if (GT[o[i]][j].c>0)
 {if (!how)
 how=s[GT[o[i]][j].y];
 else
 if (s[GT[o[i]][j].y]*how <0)
 {printf("contradictie"); return; }
 }
 else
 {if (!how)
 how=-s[GT[o[i]][j].y];
 else
 if (-s[GT[o[i]][j].y]*how<0)
 {printf("contradictie"); return; }
 }
 s[o[i]]=how;
 }
}

```

```

void write_it(void)
{int i, sum=0, sgn=1;
 FILE * fout=fopen(OutFile, "w");
 for (i=1; i<=n; i++)
 if (s[i]>0) sum++;
 if (sum<n/2) sgn=-1;
 for (i=1; i<=n; i++)
 if (sgn*s[i]<0) fprintf(fout, "0\n");
 else fprintf(fout, "1\n");
 fclose(fout); }

void top_sort()
{int i, nr=0, j, sdi[NMax];
 for (i=1; i<=n; i++) sdi[i]=di[i];
 while (nr<n)
 {for (i=1; i<=n && sdi[i]; i++)
 for (j=i+1; j<=n && sdi[j]; j++)
 o[++nr]=i; sdi[i]=-1;
 for (j=0; j<d[i]; j++) sdi[G[i][j].y]--;
 }
}

```

### Câini

Ion s-a mutat într-un sat din Alaska. Pentru a ajunge din satul său în cel mai apropiat oraș, el trebuie să traverseze un râu mare și învolburat, care nu îngheată niciodată. Ion înhamă câinii la sanie și apoi leagă de sanie o barcă. Când ajunge la râu, Ion se urcă în barcă și transportă pe rând câinii pe celălalt mal. La ultimul transport, Ion leagă sania de barcă și astfel la sfârșit ajunge pe celălalt mal cu câini, barcă și sanie. Pare destul de simplu, dar nu este întotdeauna așa. Unii câini se dușmănesc reciproc și nu pot fi lăsați singuri (fără Ion) pe același mal. Câinii care nu se dușmănesc pot fi lăsați singuri pe același mal fără nici o problemă. Evident, Ion știe care câini se dușmănesc reciproc.

De exemplu, anul acesta Ion are 5 câini: Lăbuș, Grivei, Rex, Zdreanță și Costel. Grivei se dușmănește cu Lăbuș, Zdreanță și Costel. Rex se dușmănește cu Lăbuș și Zdreanță. Celalți câini nu au aversiuni unii față de ceilalți. Prin urmare, Ion are nevoie de o barcă cu 3 locuri, astfel încât atunci când traversează râul să poată transporta doi câini și trebuie să facă cel puțin 7 traversări, după cum urmează:

1. îi transportă pe Grivei și Rex de pe malul 1 pe malul 2;
2. îl aduce înapoi pe Grivei pe malul 1;
3. îi transportă pe Costel și pe Grivei pe malul 2;
4. îi transportă pe Grivei și pe Rex înapoi pe malul 1;
5. îi transportă pe Zdreanță și pe Lăbuș pe malul 2;
6. revine pe malul 1 fără a aduce nici un câine înapoi;
7. leagă sania de barcă, îi transportă pe Grivei și pe Rex pe malul 2 și își continuă călătoria.

Observați că în nici un moment doi câini care se dușmănesc nu au fost lăsați singuri pe același mal. Dacă Lăbuș și Costel s-ar fi dușmănit, o barcă de 3 locuri nu ar mai fi fost suficientă.

#### Cerință

Scrieți un program care să determine  $x$ , numărul minim de locuri din barcă pentru a transporta toți câinii, precum și o modalitate de a transporta câinii efectuând un număr minim de transporturi cu o barcă de  $x$  locuri astfel încât, în nici un moment, doi câini care se dușmănesc să nu fie lăsați singuri pe același mal.

#### Date de intrare

Fișierul de intrare caini.in conține pe prima linie un număr natural  $N$  reprezentând numărul de câini pe care îi are Ion. Câinii sunt numerotați de la 1 la  $N$ . Pe cea de-a doua linie se află un număr natural  $K$ , reprezentând numărul de perechi de câini care se dușmănesc reciproc. Fiecare dintre următoarele  $K$  linii conține câte două numere naturale separate printr-un spațiu  $a$   $b$  cu semnificația „câinii  $a$  și  $b$  se dușmănesc reciproc”.

#### Date de ieșire

Fișierul de ieșire caini.out conține pe prima linie două numere naturale separate printr-un spațiu  $x$   $y$ ;  $x$  reprezintă numărul minim de locuri din barcă, astfel încât câinii să poată fi transportați;  $y$  reprezintă numărul minim de transporturi pe care Ion trebuie să le facă pentru a transporta toți câinii folosind o barcă de  $x$  locuri.

#### Restricții și precizări

$$1 \leq N \leq 10$$

#### Exemplu

caini.in

```
5
5
1 2
2 3
4 3
4 5
2 5
```

caini.out

```
3 7
```

(.campion, 2004)

#### Soluție

Asociem problemei un graf neorientat astfel:

- mulțimea câinilor reprezintă mulțimea vîrfurilor grafului;
  - există muchie între nodul  $a$  și nodul  $b$  în cazul în care câinii  $a$  și  $b$  se dușmănesc.
- Vom reprezenta graful prin liste de adiacență.

Numim configurație o mulțime de câini aflați pe un anumit mal. Mai exact, o configurație este o structură cu două câmpuri:

- mal care poate fi 0 sau 1;
- $T$  care este vectorul caracteristic al mulțimii câinilor situați pe malul mal.

O astfel de reprezentare este costisitoare ca memorie, prin urmare, pentru a memoria toate configurațiile corecte vom reprezenta configurațiile condensat, ca un număr (este posibil datorită numărului mic de câini). Putem considera vectorul caracteristic ca fiind reprezentarea binară a unui număr și reținem numărul respectiv în baza 10, adăugând malul drept cea mai semnificativă cifră din reprezentare.

Pentru a genera toate configurațiile de câini valide (deci în care oricare doi câini nu se dușmănesc) putem utiliza metoda *backtracking* sau un algoritm de tip succesor. Configurațiile valide generate sunt reținute în formă condensată în vectorul  $c$ .

Pe parcursul generării vom reține în variabila  $\max$  numărul maxim de câini care nu se dușmănesc, deci care pot fi lăsați singuri pe același mal. Numărul minim de locuri este  $\min_{loc} = n - \max + 1$  (dacă putem găsi o schemă de transport cu  $n - \max + 1$  locuri în barcă) sau  $\min_{loc} = n - \max + 2$  (în caz contrar).

Pentru a genera o schemă de transport a câinilor vom considera configurațiile valide ca fiind vîrfurile unui graf. Există arc de la configurația  $i$  la configurația  $j$  în acest graf dacă din configurația  $j$  se poate ajunge în configurația  $i$  cu o barcă având  $\min_{loc}$  locuri. Vom parcurge BFS acest graf plecând din configurația finală (configurația 0, în care nu există nici un câine pe malul 0, malul de plecare), până când ajungem în configurația inițială (configurația 1, în care nu există nici un câine pe malul 1). În timpul parcurgerii BFS reținem pentru fiecare configurație și numărul minim de traversări necesare pentru configurația respectivă.

```
#include <stdio.h>
#define InFile "caini.in"
#define OutFile "caini.out"
#define MAXN 14
#define DoiLa 8192
#define Inf 30000

typedef struct {int T[MAXN], mal;} Config;
FILE *out;
int c[DoiLa];
int G[MAXN][MAXN];
int n; /*numarul de caini */
int max=-1; /* numarul maxim de caini astfel incat oricare doi caini sa nu se dusmaneasca */
int nrc=0; /*numarul de configuratii din c */
int NrCaini=0;
Config S;
int Dusman[MAXN];
int ConfigToNr(Config);
void NrToConfig(int , Config *);
void Succesor(int );
void CitGraf(void);
int NrMinMutari(int );
void Afisare(int, int);

int ConfigToNr(Config S)
{int rez=0, i;
 for (i=n; i>0; i--) rez=rez*2+S.T[i];
 return rez*2+S.mal;
}
```

```

void NrToConfig(int k, Config * S)
{
    int i;
    S->mal=k%2; k/=2;
    for (i=1; i<n; i++)
        {S->T[i]= k%2;
         k/=2; }
}

void Succesor(int k)
/* cand apelam Succesor(k) in vectorul S sunt fixate
   pozitiile 0, 1, ..., k-1 */
{
    int i;
    if (k>n) /* am obtinut o multime de caini */
        {if (NrCaini>max)
            max=NrCaini;
         S.mal=0;
         /* o plasez pe malul 0, apoi pe malul 1 si retin in c
            aceste configuratii */
         c[nrc++]=ConfigToNr(S);
         S.mal=1;
         c[nrc++]=ConfigToNr(S);
        }
    else /*generam in continuare */
        {S.T[k]=0;
         Succesor(k+1);
         if (!Dusman[k])
             /*cainele k nu are dusmani in configuratia curenta*/
             {
                 for (i=1; i<=G[k][0]; i++)
                     Dusman[G[k][i]]++;
                 S.T[k]=1;
                 NrCaini++;
                 Succesor(k+1);
                 for (i=1; i<=G[k][0]; i++)
                     Dusman[G[k][i]]--;
                 NrCaini--;
             }
        }
}

void CitGraf()
{
    FILE *in=fopen(InFile,"rt");
    int K, a, b, i;
    fscanf(in,"%d %d",&n, &K);
    for (i=0; i<K; i++)
        {fscanf(in,"%d %d",&a, &b);
         G[a][0]++; G[a][G[a][0]]=b;
         G[b][0]++; G[b][G[b][0]]=a;
        }
    fclose(in);
}

```

```

int NrMinMutari(int maxloc)
/* functia returneaza numarul minim de transporturi cu o
   barca de maxloc locuri sau -1 daca nu este posibil */
{
    int Dist[Doila], Q[Doila], p, u, i, k, j, start, cati;
    Config S, S1;
    /*Dist[i]=numarul de transporturi efectuate pentru a ajunge
       din configuratia 0 in configuratia i sau -1 daca nu este
       posibil sa ajungem de la configuratia 0 la configuratia i
      */
    for (i=0; i<Doila; Dist[i++]=-1);
    Dist[0]=0;
    /* parcurgem BFS incepand din configuratia 0, configuratia
       finala, pana ajungem la configuratia 1, cea initiala */
    Q[0]=0; p=u=0;
    /*Q este coada, p indica primul element, iar u ultimul */
    while (p<=u)
        {k=Q[p++]; /*extrag prima configuratie */
         NrToConfig(k,&S);
         start=(S.mal) ? 0 : 1; /*celalalt mal */
         for (i=start; i<nrc; i+=2)
             /* parcurg toate configuratiile care corespund
                celuilalt mal */
             {int k1=c[i];
              NrToConfig(k1,&S1);
              if (Dist[k1]==-1)
                  {
                      /* verific daca se poate ajunge de la configuratia
                         k1 la configuratia k intr-un singur transport,
                         folosind o barca cu maxloc locuri */
                      cati=0;
                      /*numar cati caini au fost transportati de pe
                         malul configuratiei k1 pe malul configuratiei k*/
                      for (j=1; j<=n; j++)
                          if (S1.T[j])
                              {if (S.T[j])
                                 {cati=Inf; break; } }
                           else
                               if (!S.T[j]) cati++;
                      if (cati<=maxloc)
                          /*incap in barca cainii transportati? */
                          {Dist[k1]=Dist[k]+1;
                           if (k1==1) return Dist[k1];
                           /*am transportat toti cainii */
                           Q[++u]=k1;
                          }
                      }
                  }
        }
    return -1;
}

```

```

void Afisare(int minmut, int minloc)
{
    int temp[DoiLa], i, j;
    FILE *out=fopen(OutFile,"wt");
    fprintf(out,"%d %d\n", minloc, minmut);
    fclose(out);
}

int main()
{
    int t;
    CitGraf();
    /*generez toate configuratiile de caini care nu se ataca */
    Succesor(1);
    t=NrMinMutari(n-max);
    if (t!= -1) Afisare(t, n-max+1);
    else Afisare(NrMinMutari(n-max+1), n-max+2);
    return 0;
}

```

### Ghizi

Se caută ghizi pentru Olimpiada Națională de Informatică. Deoarece la Olimpiadă participă  $K$  echipe, trebuie ca în fiecare moment de timp din intervalul  $[0, 100]$  să existe exact  $K$  ghizi.

Pentru posturile de ghid s-au înscris  $N$  voluntari, care au fost numerotăți distinct de la 1 la  $N$ . Fiecare dintre cei  $N$  voluntari a specificat un interval de timp în care poate asigura servicii de ghid. Volunteerul  $i$  poate fi ghid în intervalul  $[T_{1i}, T_{2i}]$ .

### Cerință

Dându-se intervalele de timp asociate celor  $N$  voluntari, determinați o variantă de angajare astfel încât în fiecare moment de timp, să fie prezenți exact  $K$  ghizi. Numărul total de voluntari angajați este irrelevant.

### Date de intrare

Prima linie a fișierului de intrare `ghizi.in` conține două numere întregi  $N$  și  $K$ , separate printr-un spațiu, cu semnificațiile din enunț. Fiecare dintre următoarele  $N$  linii conține descrierea unui volunteer; mai exact, linia  $i+1$  conține valorile întregi  $T_{1i}$  și  $T_{2i}$  pentru voluntarul  $i$ .

### Date de ieșire

În fișierul `ghizi.out` veți afișa pe prima linie numărul total de ghizi angajați  $M$ . Pe a doua linie veți scrie  $M$  numere distincte între 1 și  $N$ , ordonate crescător, reprezentând numerele de ordine ale ghizilor angajați.

### Restricții și precizări

$1 \leq N \leq 5000$

$0 \leq T_1 < T_2 \leq 100$  pentru fiecare dintre cei  $N$  voluntari

$1 \leq K \leq N$

Pot exista 2 voluntari cu același interval asociat.

Toate testeile vor avea soluție (dacă există mai multe soluții, afișați una oricare).

### Exemplu

ghizi.in	ghizi.out
6 2	4
0 100	1 4 5 6
0 15	
15 99	
0 10	
10 20	
20 100	

(Olimpiada Națională de Informatică, baraj, 2003)

### Soluție

Asociem problemei un graf orientat astfel:

- fiecare moment întreg de timp din intervalul  $[0, 100]$  este un vârf în graf;
- pentru fiecare interval de timp  $[T_{1i}, T_{2i}]$  asociat unui voluntar introducem un arc în graf de la vârful  $T_{1i}$  la vârful  $T_{2i}$ .

Asociem fiecarui arc din graf capacitatea 1 și transformăm astfel graful într-o rețea de transport (vârful 0 fiind sursa, iar vârful 100 fiind destinația).

Orice drum de la sursă la destinație asigură în fiecare moment de timp existența unui singur ghid.

Prin urmare, putem să reformulăm problema astfel: să se determine  $K$  drumuri de la sursă la destinație, disjuncte din punctul de vedere al arcelor.

Vom introduce în graf un nod suplimentar, nodul 101, și vom introduce un arc de la vârful 101 la vârful 0, capacitatea acestui arc fiind  $K$ . Vârful 101 devine sursa în rețea de transport. Determinând un flux maxim în rețea, determinăm numărul maxim de drumuri de la sursă la destinație, drumuri disjuncte din punctul de vedere al arcelor.

Din precizarea că pot exista doi voluntari cu același interval asociat deducem că graful asociat problemei este de fapt un multigraf. Rezolvăm această problemă astfel: capacitatea arcului  $(i, j)$  este egală cu numărul de voluntari care au intervalul de timp asociat  $[i, j]$ .

O altă problemă va apărea la afișare: va trebui să identificăm voluntarii asociați arcelor pentru care fluxul este nenul. Din acest motiv vom defini structura arc cu trei câmpuri (extremitățile arcului și câmpul uz, în care reținem valoarea 1 dacă voluntarul corespunzător este utilizat în soluție și 0, în caz contrar). Pentru a identifica voluntarii care constituie soluția, pentru fiecare arc cu fluxul nenul, vom selecta pentru fiecare unitate de flux căte un voluntar care are un interval de timp corespunzător cu acel arc.

O ultimă problemă ar apărea din cauza faptului că primul moment de timp este 0. Indexând vâfurile grafului de la 0 vom avea probleme cu marcarea vâfurilor în timpul parcurgerii în lățime. Din acest motiv vom incrementa fiecare moment de timp cu o unitate (deci arcele grafului vor fi numerotate de la 1 la 102).

```

#include <stdio.h>
#include <stdlib.h>
#define MAXV 103
#define MAXN 5001
#define min(x,y) ((x)<(y)?(x):(y))
#define abs(x) ((x)>0?x:-x)

int n, k, s, d, viz[MAXV], Q[MAXV];
int *C[MAXV]; //capacitatea fiecarui arc
int *F[MAXV]; //fluxul fiecarui arc
struct arc {int x, y, uz;} G[MAXN];

void citire(void);
void ek(void);
void afisare(void);
int bfs(void);

int main()
{
    citire();
    ek();
    afisare();
    return 0;
}

void citire()
{
    FILE * fin=fopen("ghizi.in","r");
    int i, t1, t2;
    for (i=0; i<MAXV; i++)
        C[i]=(int *) calloc(MAXV, sizeof(int));
    F[i]=(int *) calloc(MAXV, sizeof(int));
    fscanf(fin, "%d %d", &n, &k);
    for (i=1; i<=n; i++)
        fscanf(fin, "%d %d", &t1, &t2);
    C[t1+1][t2+1]++;
    G[i].x=t1+1; G[i].y=t2+1;
    C[MAXV-1][1]=k;
    s=MAXV-1; d=MAXV-2;
}

void afisare()
{
    int i, j, p, nr=0;
    FILE * fout=fopen("ghizi.out", "w");
    for (i=1; i<MAXV-1; i++)
        for (j=1; j<MAXV-1; j++)
            if (F[i][j])
                for (p=1; p<=n; p++)
                    if (G[p].x==i && G[p].y==j && G[p].uz==0)
                        G[p].uz=1; nr++; break;
    fprintf(fout, "%d\n", nr);
    for (i=1; i<=n; i++)
        if (G[i].uz) fprintf (fout, "%d ", i);
    fclose(fout);
}

```

```

void ek()
{
    int i, a, b, lg, v;
    int L[MAXV];
    do
    {
        for (i=1; i<MAXV; i++) viz[i]=0;
        if (bfs()) return;
        //determinam lantul in vectorul L
        L[0]=d; lg=0;
        a=10000;
        while (L[lg]!=s)
            L[++lg]=abs(viz[L[lg-1]]);
        if (viz[L[lg]]>0)
            a=min(a,C[L[lg]][L[lg-1]]-F[L[lg]][L[lg-1]]);
    }
    //marim fluxul de-a lungul lantului
    for (i=lg; i>0; i--) F[L[i]][L[i-1]]+=a;
}
while (1);

int bfs()
//returneaza 1 daca iesirea retelei nu a fost marcata
{
    int p, u, i, x;
    Q[0]=s; p=u=0; viz[s]=1;
    while (p<=u && !viz[d])
        (x=Q[p++]);
        for (i=1; i<MAXV; i++)
            if (!viz[i])
                if (F[x][i]<C[x][i])
                    {viz[i]=x; Q[++u]=i;}
    }
    return !viz[d];
}

Auto

Eu locuiesc în orașul Iași (să-l notăm pentru simplitate orașul A) și trebuie să ajung în cursus de maxim T ore în București (să-l notăm orașul B). Eu nu circul decât pe autostradă, cu viteza maximă legală, aşa că am luat o hartă pe care sunt marcate cele N orașe din țară, precum și cele M autostrăzi ce realizează legătura dintre orașe. Fiecare autostradă face legătura între două orașe. Harta este foarte detaliată și prezintă și taxele de autostradă și costul parcerii în fiecare oraș.

În orașul meu (A) și în orașul destinație (B) eu nu plătesc parcare, dar, dacă vreau să opresc în orice alt oraș, trebuie să plătesc o taxă de parcare (de exemplu, pentru orașul i, taxa de parcare va fi  $p_i$  RON/oră).

Taxele de autostradă diferă în funcție de momentul în care intru pe autostradă. Taxa se plătește pentru fiecare oră de mers pe autostradă, deci, dacă am mers pe o autostradă h ore și la momentul intrării pe autostradă taxa era c, la ieșirea de pe autostradă voi plăti  $c * h$  RON.
```

**Cerință**

Scrieți un program care să determine suma totală minimă (care reprezintă cost parcare + cost taxe de autostradă) pe care trebuie să o plătesc pentru a ajunge din orașul A în orașul B în maxim T ore.

**Date de intrare**

Fișierul de intrare `auto.in` conține pe prima linie două numere naturale separate printr-un spațiu  $N \ M$ , reprezentând numărul de orașe, respectiv, numărul de autostrăzi. Pe cea de-a doua linie se află trei numere separate prin spațiu  $A \ B \ T$ , reprezentând orașul de plecare, orașul destinație și timpul maxim în care trebuie să ajung la destinație. Pe a treia linie se află  $N$  numere naturale separate prin căte un spațiu  $p_1 \ p_2 \dots \ p_n$ , reprezentând costul de parcare/oră în fiecare dintre cele  $N$  orașe. Pe următoarele  $2 \times M$  linii se află informații despre cele  $M$  autostrăzi, căte două linii pentru o autostradă. Pe prima linie dintre cele două sunt scrise trei numere naturale separate prin căte un spațiu  $O_1 \ O_2 \ D$ , cu semnificația „autostrada leagă orașele  $O_1$  și  $O_2$  și parcurgerea ei durează  $D$  ore la viteza maximă legală”. Pe cea de-a doua linie sunt scrise  $T$  numere naturale separate prin căte un spațiu  $c_0 \ c_1 \dots \ c_{T-1}$ , unde  $c_i$  reprezintă costul pentru o oră de mers pe autostrada respectivă, dacă am intrat la momentul  $i$ .

**Date de ieșire**

Fișierul de ieșire `auto.out` va conține o singură linie pe care va fi scris un număr natural, reprezentând suma totală minimă pe care trebuie să o plătesc pentru a ajunge din orașul A în orașul B în timpul dat.

**Restricții și precizări**

$0 < N, T \leq 100$

$M \leq 500$

$0 \leq p_i, c_i \leq 100$

Eu plec din orașul A la momentul 0.

Între două orașe există cel mult o autostradă.

**Exemplu**

auto.in	auto.out
3 2	7
1 3 5	
0 1 0	
1 2 2	
2 5 5 5 5	
2 3 2	
5 5 5 1 5	

**Explicație**

Plec din orașul 1 la momentul 0, merg pe autostrada 1 2 timp de 2 ore (deci plătesc  $2 \times 2 = 4$  RON). Am ajuns în orașul 2 la momentul 2. Parchez aici o oră și plătesc 1 RON. Apoi intru pe autostrada 2 3 pe care merg 2 ore (deci plătesc  $2 \times 1$  RON). În total 7 RON.

(.campion, 2005)

**Soluție**

Putem asocia problemei un graf neorientat astfel :

- vârfurile grafului sunt cele  $N$  orașe;
- dacă există autostradă între orașele  $i$  și  $j$ , plasăm în graf o muchie între  $i$  și  $j$ .

**Reprezentarea informațiilor :**

1. În vectorul  $p$  vom reține costurile de parcare  $p[i] =$  costul unei ore de parcare în orașul  $i$ .

2. Graful va fi reprezentat prin matricea de adiacență  $mad$ .
3. În vectorul  $lung$  vom reține lungimea autostrăzilor:  $lung[k] =$  numărul de ore necesare pentru a parcurge autostrada  $k$ .
4. Costurile de autostradă le reținem în matricea `cost_auto`:  $cost\_auto[k][t] =$  taxa pentru o oră de mers pe autostrada  $k$ , dacă intrăm la ora  $t$ .
5. Pentru a determina soluția problemei vom utiliza o matrice `cmin` cu  $N$  linii și  $T$  coloane cu semnificația:  $cmin[j][i] =$  costul total minim pentru a ajunge în orașul  $j$  în  $i$  ore. Soluția problemei va fi minimul de pe linia  $B$ .

Rezolvăm problema prin programare dinamică.

Pentru a ajunge în orașul  $j$  în  $i$  ore, putem merge în orașul 1 ( $i=0, n-1$ ) în  $k$  ( $k=0, tf$ ) ore, apoi putem parcurge autostrada de la orașul 1 la orașul  $j$  (dacă aceasta există este autostrada  $mad[1][j]$ ).

Să notăm cu  $lm =$  numărul de ore necesar pentru a parcurge drumul de la 1 la  $j$  ( $lm=lung[mad[1][j]]$ ). Pe autostrada  $mad[1][j]$  vom intra la momentul  $i-lm$ , deci taxele de autostrada vor fi  $cost\_auto[mad[1][j]][i-lm]*lm$ .

Costul total care se obține în acest mod este: costul pentru a ajunge în orașul 1 în  $k$  ore ( $cmin[1][k]$ ) + taxele pe autostradă de la 1 la  $j$  ( $cost\_auto[mad[1][j]][i-lm]*lm$ , dacă  $i-lm>0$ ) + costul parcerii în orașul 1 timp de ( $i-k-lm$ ) ore (deci  $p[1]*(i-k-lm)$ , dacă  $i-k \geq lm$ ). Adică:

$$c\_min[1][k]+cost\_auto[mad[1][j]][i-lm]*lm+p[1]*(i-k-lm)$$

Evident,  $cmin=\min(c\_min[1][k]+cost\_auto[mad[1][j]][i-lm]*lm+p[1]*(i-k-lm) | k=0, tf; i=0, n-1)$

```
#include <stdio.h>
#define InFile "auto.in"
#define OutFile "auto.out"
#define NMax 101
#define MMax 500
#define TMax 101
#define oo 300000000

int n, m, a, b, tf;
int p[NMax];
int mad[NMax][NMax];
int lung[MMax];
int cost_auto[MMax][TMax];
int c_min[NMax][TMax];
void Read_it(void);
void Solve(void);
void Write_it(void);

int main()
{Read_it();
 Solve();
 Write_it();
 return 0;
}
```

```

void Read_it()
{
    int i, j, o1, o2, l;
    FILE * fin=fopen(InFile, "r");
    fscanf(fin, "%d %d\n", &n, &m);
    fscanf(fin, "%d %d %d\n", &a, &b, &tf);
    a--; b--;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            mad[i][j]=-1;
    for (i=0; i<n; i++)
        fscanf(fin, "%d", &p[i]);
    p[a]=p[b]=0;
    for (i=0; i<m; i++)
        fscanf(fin, "%d %d %d\n", &o1, &o2, &l);
        mad[o1-1][o2-1]=mad[o2-1][o1-1]=i;
        lung[i]=l;
        for (j=0; j<tf; j++)
            fscanf(fin, "%d", &cost_auto[i][j]);
    }
    fclose(fin);
}

void Solve()
{
    int i, j, k, l, lm, m;
    for (i=0; i<n; i++)
        for (j=0; j<tf; j++) c_min[i][j]=-1;
    c_min[a][0]=0;
    for (i=1; i<tf; i++)
        for (j=0; j<n; j++)
            if (c_min[j][i]==oo)
                for (k=0; k<i; k++)
                    for (l=0; l<n; l++)
                        if (c_min[l][k]==-1) continue;
                        if (mad[j][l]==-1) continue;
                        lm=lung[mad[j][l]];
                        if (lm>i-k) continue;
    m=c_min[1][k]+cost_auto[mad[j][l]][i-lm]*lm+p[l]*(i-k-lm);
    if (m<c_min[j][i]) c_min[j][i]=m;
    if (c_min[j][i]==oo) c_min[j][i]=-1;
}

void Write_it()
{
    FILE * fout=fopen(OutFile, "w");
    int j, min=oo;
    for (j=0; j<tf; j++)
        if (c_min[b][j]!=-1 && c_min[b][j]<min)
            min=c_min[b][j];
    fprintf(fout, "%d\n", min);
    fclose(fout);
}

```

### Lanterna

Un agent secret are o hartă pe care sunt marcate  $N$  obiective militare. El se află, inițial, lângă obiectivul numerotat cu 1 (baza militară proprie) și trebuie să ajungă la obiectivul numerotat cu  $N$  (baza militară inamică). În acest scop, el va folosi drumurile existente, fiecare drum legând două obiective distincte. Fiind o misiune secretă, deplasarea agentului va avea loc noaptea; de aceea, el are nevoie de o lanteră. Pentru aceasta, el are de ales între  $K$  tipuri de lanterne – o lanternă de tipul  $w$  ( $1 \leq w \leq K$ ) are baterii care permit consumul a  $w$  wați, după consumul acestor wați, lanterna nu mai luminează. Din fericire, unele dintre obiective sunt baze militare prietene, astfel că, odată ajuns acolo, el își poate reîncărca bateriile complet. Agentul trebuie să aibă grijă ca, înainte de a merge pe un drum între două obiective, cantitatea de wați pe care o mai poate consuma să fie mai mare sau egală cu cantitatea de wați pe care o va consuma pe drumul respectiv.

Cunoscând drumurile dintre obiective și, pentru fiecare drum, durata necesară parcurgerii drumului și numărul de wați consumați de lanternă, determinați tipul de lanternă cu numărul cel mai mic, astfel încât durata deplasării să fie minimă (dintre toate tipurile de lanternă cu care se poate ajunge în timp minim la destinație, interesează lanterna cu consumul cel mai mic).

#### Date de intrare

Pe prima linie a fișierului lanterna.in se află numerele întregi  $N$  și  $K$ , separate printr-un spațiu. Pe următoarea linie se află  $N$  numere întregi din mulțimea {0, 1}. Dacă al  $i$ -lea număr este 1, aceasta înseamnă că obiectivul cu numărul  $i$  este o bază militară prietenă (adică agentul își poate reîncărca bateriile lanternei dacă ajunge la acest obiectiv); dacă numărul este 0, agentul nu își va putea reîncărca bateriile. Primul număr din linie este 1, iar ultimul este 0. Pe cea de-a treia linie a fișierului se află numărul  $m$  de drumuri dintre obiective. Fiecare dintre următoarele  $m$  linii conține câte 4 numere întregi separate prin spații:  $a$   $b$   $T$   $w$ , având semnificația că există un drum bidirectional între obiectivele  $a$  și  $b$  ( $a \neq b$ ), care poate fi parcurs într-un timp  $T$  și cu un consum de  $w$  wați.

#### Date de ieșire

În fișierul lanterna.out se vor afișa două numere întregi, separate printr-un spațiu:  $T_{\min}$  și  $w_{\min}$ .  $T_{\min}$  reprezintă durata minimă posibilă a deplasării de la obiectivul 1 la obiectivul  $N$ , iar  $w_{\min}$  reprezintă tipul de lanternă cu numărul cel mai mic pentru care se obține acest timp.

#### Restricții și precizări

$2 \leq N \leq 50$   
 $1 \leq K \leq 1000$   
 $1 \leq m \leq N^*(N-1)/2$

Între două obiective diferite poate exista maxim un drum direct.

Pentru fiecare drum, durata parcurgerii este un număr întreg cuprins între 1 și 100, iar numărul de wați consumați este un număr întreg între 0 și 1000.

Se garantează că există cel puțin un tip de lanternă pentru care deplasarea este posibilă.

**Exemplu**

```
lanterna.in
7 10
1 0 1 0 0 0 0
7
1 2 10 3
1 4 5 5
2 3 10 3
4 3 15 1
3 6 4 3
6 5 2 2
5 7 1 0
```

```
lanterna.out
27 6
```

(Olimpiada Județeană de Informatică, 2004)

**Soluție**

Asociem problemei un graf neorientat astfel :

- vârfurile grafului sunt cele N obiective ;
  - există muchia  $[i, j]$  dacă și numai dacă există drum între obiectivele i și j.
- Fiecare muchie din graf are asociate două valori : timpul necesar parcurgerii drumului corespunzător muchiei și consumul de wați necesar.

Problema cere să determinăm costul lanțului de cost minim de la vârful 1 la vârful N. Costul unui lanț este egal cu suma timpilor asociați muchiilor din care este constituit lanțul. De asemenea, se solicită și tipul minim de lanternă, cu care se poate ajunge de la vârful 1 la vârful N în timpul minim determinat.

Pentru a determina timpul minim de a ajunge de la obiectivul 1 la obiectivul N folosind o lanternă cu puterea kmax vom utiliza funcția MinT. Pentru a calcula acest timp minim vom utiliza o coadă în care vom reține configurațiile curente. Prin configurație înțelegem un vârf din graf împreună cu consumul necesar pentru a ajunge într-un timp minim în vârful respectiv.

Inițial suntem în obiectivul 1 și am consumat 0 wați, deci vom inițializa coada cu configurația  $(1, 0)$ . Cât timp coada nu este vidă, extragem din coadă o configurație  $(i, k)$  și „expandăm” această configurație.

Mai exact, pentru toate vârfurile j adiacente cu vârful i, în care putem ajunge, adică vârfuri pentru care  $k + W[i][j] \leq k_{\max}$  (consumul curent) +  $W[i][j]$  (consumul pentru a ajunge de la i la j) ≤  $k_{\max}$  (puterea lanternei), determinăm consumul q cu care ajungem în j (acesta este 0 dacă j este obiectiv prieten sau  $k + W[i][j]$ ) și reținem timpul în care am ajuns,  $T_{min}[i][k] + T[i][j]$  (dacă acesta este  $< T_{min}[j][q]$ ). Apoi adăugăm în coadă configurația  $(j, q)$ . Pentru a nu adăuga aceeași configurație în coadă de mai multe ori vom utiliza o matrice  $InC$  cu N linii și  $k_{\max}$  coloane ( $InC[i][k] = 1$  dacă a fost inserată în coadă configurația  $(i, k)$ , respectiv 0, în caz contrar). În timpul parcurgerii calculăm în variabila tN timpul minim necesar pentru a ajunge cu o lanternă de putere  $k_{\max}$  în obiectivul N.

Pentru a determina timpul minim  $minTimp$ , în care putem ajunge de la obiectivul 1 la obiectivul N, apelăm funcția  $MinT(K)$  (unde K era cea mai mare putere posibilă pentru lanternă).

Pentru a determina puterea minimă a lanternei cu care putem ajunge la obiectivul N în timp minim vom face o căutare binară după puterea lanternei. Să notăm cu  $l_1$ ,  $l_s$  capetele intervalului în care căutăm binar puterea optimă a lanternei (inițial  $l_1=1$ ,  $l_s=k-1$ ). Determinăm  $p=(l_1+l_s)/2$  mijlocul intervalului.

Dacă  $MinT(p)$  (adică timpul minim necesar pentru a ajunge în obiectivul N) este egal cu  $minTimp$ , atunci p este o putere mai bună decât cea curentă, o reținem și căutăm în continuare în intervalul  $[l_1, p-1]$ . Dacă  $MinT(p) > minTimp$ , deducem că această putere pentru lanternă este prea mică, vom continua căutarea în intervalul  $[p+1, l_s]$ .

```
#include <iostream.h>
#define MAXN 51
#define MAXK 1001
#define DMax 50000
#define infinit 30000

typedef struct {
    char i; //varful
    int k; //consumul cu care s-a ajuns in varful i
} ElemtCoada;

ElemtCoada *C;

int T[MAXN][MAXN], W[MAXN][MAXN];
int *Tmin[MAXN];
int prieten[MAXN];
char InC[MAXN][MAXK];
int N, K, minTimp, minK;

void citire()
{
    ifstream fin("lanterna.in");
    int i, j, p, M;
    fin>>N>>K;
    for (i=1; i<=N; i++) fin>>prieten[i];
    fin>>M;
    for (i=1; i<=N; i++)
        for (j=1; j<=N; j++)
            {T[i][j]=infinit;
             W[i][j]=MAXK+1; }
    for (p=1; p<=M; p++)
        {fin>>i>>j;
         fin>>T[i][j]>>W[i][j];
         T[j][i]=T[i][j];
         W[j][i]=W[i][j];
        }
    //alocam memorie dinamic pentru coada C si matricea Tmin
    for (i=0; i<=N; i++)
        Tmin[i]=new int [K+1];
    C=new ElemtCoada[DMax];
}
```

```

void afisare()
{
    ofstream fout("lanterna.out");
    fout<<minTimp<< ' '<<minK<<endl;
    fout.close();
}

int MinT(int kmax)
/* returneaza timpul minim in care se ajunge de la vf 1 la
vf N cu o lanterna de tipul kmax */
{
    int i, j, k, p, q, tN, vf, u;
    for (i=1; i<=N; i++)
        for (j=0; j<=kmax; j++)
            Tmin[i][j]=infinit;
    Tmin[1][0]=0; InC[1][0]=1;
    C[0].i=1; C[0].k=0;
    vf=u=0; tN=infinit;
    while (vf<=u)
        {
            i=C[vf].i; k=C[vf++].k;
            if (Tmin[i][k]<tN)
                for (j=1; j<=N; j++)
                    if (T[i][j]<infinit && k+W[i][j]<=kmax)
                        {p=Tmin[i][k]+T[i][j];
                         q=k+W[i][j];
                         if (prieten[j]) q=0;
                         if (p<Tmin[j][q])
                             {Tmin[j][q]=p;
                              if (j==N && Tmin[j][q]<tN)
                                  tN=Tmin[j][q];
                              if (!InC[j][q] && Tmin[j][q]<tN)
                                  {InC[j][q]=1;
                                   C[++u].i=j; C[u].k=q;
                                   }
                             }
                         }
            InC[i][k]=0;
        }
    return tN;
}

int main()
{
    int li=1, ls=K-1, p;
    citire();
    minTimp=MinT(K);
    minK=K;
    while (li<=ls)
        { p=(li+ls)>>1;
          if (MinT(p)==minTimp)
              {minK=p;
               ls=p-1;}
          else li=p+1;
        }
    afisare();
    return 0;
}

```

*Walls*

Într-o țară s-au construit niște ziduri astfel încât orice zid conectează exact două orașe. Zidurile nu se intersectează. Prin urmare, țara este împărțită în regiuni astfel încât, pentru a trece de la o regiune la alta, trebuie să fie traversat un zid sau un oraș. Pentru oricare două orașe A, B există cel mult un zid cu o extremitate în A și cealaltă în B; în plus, pentru a ajunge de la orașul A la orașul B trebuie să se meargă de-a lungul zidurilor sau să se traverseze alte orașe.

În această țară există un club al cărui membri locuiesc în orașe. În fiecare oraș există cel mult un membru al acestui club. Membrii vor să se întâlnească într-o regiune (în afara oricărui oraș). Ei vor veni la întâlnire cu bicicleta. Din cauza traficului, ei nu vor să intre în nici un oraș și vor să traverseze cât mai puține ziduri.

Fiecare membru va traversa până la locul de întâlnire un anumit număr de ziduri (posibil 0). O regiune este considerată loc optim de întâlnire dacă numărul total de ziduri traversate de membrii clubului este minim.

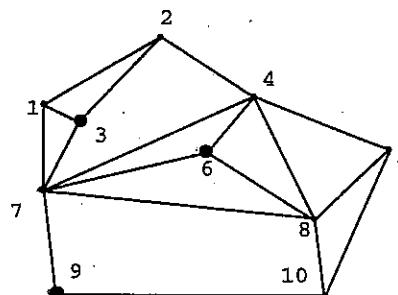


Figura 1

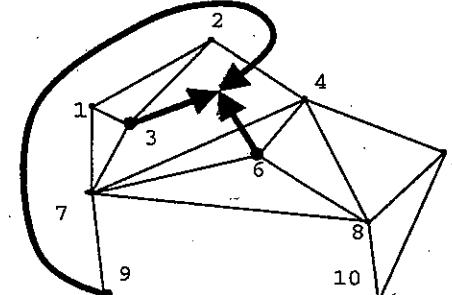


Figura 2

Orașele sunt identificate prin numere de la 1 la N.

În figura 1, punctele etichetate reprezintă orașe, iar liniile reprezintă ziduri. Să presupunem că există 3 membri care locuiesc în orașele 3, 6, și 9. Atunci locul optim de întâlnire și traseele posibile sunt ilustrate în figura 2. Numărul total de ziduri traversate este 2 (membrul care locuiește în orașul 9 trebuie să traverseze zidul dintre orașele 2 și 4, iar membrul din orașul 6 trebuie să traverseze zidul dintre orașele 4 și 7).

*Cerință*

Scrieți un program care să determine un loc optim de întâlnire, precum și numărul total minim de ziduri care trebuie să fie traversate de membrii clubului până la locul de întâlnire.

*Date de intrare*

Fișierul de intrare walls.in conține pe prima linie un număr natural m, reprezentând numărul de regiuni.

Pe cea de-a două linie se află numărul natural  $N$ , reprezentând numărul de orașe. Pe cea de-a treia linie se află un număr natural  $L$ , reprezentând numărul de membri ai clubului. Pe cea de-a patra linie se află o secvență crescătoare de etichete, reprezentând orașele în care locuiesc membrii clubului.

Urmează în fișier  $2M$  linii, câte două linii pentru fiecare regiune. Prima dintre cele două linii conține numărul de orașe  $I$  și aflete pe frontieră regiunii corespunzătoare perechii de linii. Cea de-a două linie conține cele  $I$  orașe de pe frontieră, în ordine, astfel încât pot fi parcuse în sensul acelor de ceas, cu o singură excepție.

Ultima regiune este regiunea exterioară, cea care înconjoară toate celelalte regiuni și orașe. Pentru această ultimă regiune, orașele de pe frontieră sunt specificate în ordinea inversă a celor de ceas. Regiunile sunt numerotate de la 1 la  $M$ , conform ordinii din fișierul de intrare.

#### Date de ieșire

Fișierul de ieșire `walls.out` va conține pe prima linie un număr natural reprezentând numărul total minim de traversări de ziduri până la locul de întâlnire. Pe cea de-a două linie se va afla regiunea în care se găsește locul optim de întâlnire.

#### Restricții

$2 \leq M \leq 200$   
 $3 \leq N \leq 250$   
 $1 \leq L \leq 30$ ,  $L \leq N$

#### Exemplu

`walls.in`

```
10
10
3
3 6 9
3
1 2 3
3
1 3 7
4
2 4 7 3
3
4 6 7
3
4 8 6
3
6 8 7
3
4 5 8
4
7 8 10 9
3
5 10 8
7
7 9 10 5 4 2 1
```

`walls.out`

```
2
3
```

(Olimpiada Internațională de Informatică, 2000)

#### Soluție

Asociem problemei un graf neorientat astfel :

- vîrfurile grafului sunt regiunile ;
- dacă două regiuni au un zid comun, atunci vîrfurile corespunzătoare lor vor fi adiacente.

Pentru a construi graful observăm că fiecare zid apare exact de două ori în datele de intrare (o dată parcurs într-un sens, o dată în celălalt sens).

Vom reprezenta graful prin matricea de adiacență  $A$ . În plus, vom utiliza o matrice specială  $G$ , în care pentru oricare două orașe  $i$ ,  $j$  vom reține :  $G[i][j] =$  numărul regiunii din care face parte zidul  $[i, j]$ , parcurs în acest sens. Când citim orașele de pe frontieră regiunii  $k$ , vom reține în matricea  $G$  pentru fiecare zid de pe frontieră orașul  $k$ .

Pentru a reține orașele în care locuiesc membrii clubului vom utiliza un vector  $o$  ;  $o[i]=0$ , dacă în orașul  $i$  nu locuiește nici un membru al clubului ; sau  $o[i]=$  numărul de ordine în lista de orașe al orașului  $i$ . Pentru a reține pentru fiecare oraș din lista de orașe care sunt regiunile pe care le-a căror frontieră se află orașul, vom utiliza o matrice  $AD$  cu  $30$  de linii și  $250$  de coloane.  $AD[i][j]=1$ , dacă al  $i$ -lea oraș din listă apare pe frontieră regiunii  $j$  și  $0$ , în caz contrar.

Pentru a calcula numărul total minim de ziduri care trebuie să fie traversate vom calcula distanțele de la fiecare oraș în care locuiește un membru al clubului la fiecare regiune (distanța este calculată în număr de ziduri traversate). Distanțele vor fi reținute în matricea  $D$  ( $D[i][j] =$  distanța de la al  $i$ -lea oraș din listă la regiunea  $j$ ). Pentru a calcula distanțele este suficient să realizăm o parcurgere în lățime în graful asociat problemei începând din orașul în care locuiește membrul respectiv. Mai exact (fiindcă vîrfurile grafului sunt regiuni, nu orașe), vom inițializa coada corespunzătoare parcurgerii BFS cu toate regiunile care conțin pe frontieră orașul în care locuiește membrul clubului.

La final vom calcula pentru fiecare regiune numărul total de traversări de ziduri și vom alege minimul.

```
#include <stdio.h>
#define NMax 251
#define MMax 201
#define OMax 31

int N, M, L;
char G[NMax][NMax];
char A[MMax][MMax];
int o[NMax];
char AD[OMax][MMax];
int D[OMax][MMax];
char viz[MMax];
int C[MMax]; //coada
long Min;
int R;
```

```

void Citire()
{
    FILE * fin=fopen("walls.in","r");
    int nr, x, y, j, i, prim;
    fscanf(fin,"%d %d %d", &M, &N, &L);
    for (i=1; i<=L; i++)
        fscanf(fin, "%d", &x); o[x]=i;
    for (i=1; i<=M; i++)
        fscanf(fin, "%d %d", &nr, &prim);
        x=prim;
        for (j=1; j<=nr; x=y, j++)
            if (j<nr) fscanf(fin, "%d", &y);
            else y=prim;
            G[x][y]=i;
            if (o[y]) AD[o[y]][i]=1;
            A[i][G[y][x]]=A[G[y][x]][i]=1;
        }
    }

void BFS(int start)
{
    int i, prim=0, ultim=-1, x;
    for (i=1; i<=M; i++) viz[i]=0;
    for (i=1; i<=M; i++)
        if (AD[start][i])
            {C[++ultim]=i; viz[i]=1;
             D[start][i]=0;}
    while (prim<=ultim)
        {x=C[prim++];
         for (i=1; i<=M; i++)
             if (!viz[i] && A[i][x])
                 {viz[i]=1;
                  C[++ultim]=i;
                  D[start][i]=1+D[start][x];
                 }
        }
    }

void Rezolvare()
{
    int i, j;
    long s;
    for (i=1; i<=L; i++) BFS(i);
    for (Min=1000000, i=1; i<=M; i++)
        {for (s=0, j=1; j<=L; j++) s+=D[j][i];
         if (s<Min) {Min=s; R=i,}
        }
    }

void Afisare()
{
    FILE * fout=fopen("walls.out","w");
    fprintf(fout,"%ld\n%ld\n", Min, R);
    fclose(fout);
}

```

```

int main()
{
    Citire();
    Rezolvare();
    Afisare();
    return 0;
}

```

### Jungla

*ABC Pharmaceuticals* și *XYZ Pharmaceuticals* au finanțat o expediție științifică în junglă. În această expediție a fost explorată o zonă albă de pe harta lumii, un loc unde „mâna omului n-a pus niciodată piciorul”.

În junglă, exploratorii au găsit N triburi, pe care le-au numerotat 1, 2, ..., N. Unele dintre aceste triburi comunică, prin urmare, exploratorii au construit o hartă pe care au marcat poziția fiecărui trib și drumurile existente între pozițiile triburilor. Cea mai interesantă descoperire este că aceste triburi au cunoștințe extraordinare despre cum pot fi folosite plantele care cresc în zona lor pentru tratarea diferitelor boli.

Bill este un angajat al *ABC Pharmaceuticals*, iar John este un angajat al *XYZ Pharmaceuticals*. De curând, ei au primit o misiune grea: trebuie să meargă în junglă și să încheie contracte de colaborare exclusivă între triburi și firmele lor. În acest scop, ei vor efectua împreună mai multe călătorii.

Pentru prima călătorie, ei au la dispoziție un elicopter care îi va duce în junglă la unul dintre triburi și îi va aștepta acolo. Pentru a vizita alte triburi, Bill și John vor călători pe jos, utilizând numai drumurile marcate pe hartă. Bill și John consideră că prima călătorie este convenabilă dacă respectă următoarele condiții:

- ei vizitează un număr par de triburi astfel încât Bill va încheia un contract cu primul trib, John cu cel de-al doilea, Bill cu cel de al treilea etc.
- nu vizitează același trib de mai multe ori (exceptând primul trib, cel de la care pleacă);
- numărul de triburi vizitate este minim;
- și, bineînteles, pot vizita triburile mergând numai pe drumurile marcate pe hartă și să revină la poziția în care îi așteaptă elicopterul.

### Cerință

Scrieți un program care să determine o călătorie convenabilă pentru Bill și John.

#### Date de intrare

Fișierul de intrare *jungla.in* conține:

- pe prima linie două numere naturale N M, separate prin spațiu, reprezentând numărul de triburi și respectiv numărul de drumuri directe existente între triburi;
- fiecare dintre următoarele M linii conține două numere naturale x y, separate prin spațiu, cu semnificația „între tribul x și tribul y există un drum direct”.

#### Date de ieșire

Fișierul de ieșire *jungla.out* conține pe prima linie un număr natural par P, reprezentând numărul de triburi vizitate (minim). Pe cea de-a doua linie se află cele P triburi vizitate, scrise în ordinea vizitării, oricare două triburi consecutive fiind separate printr-un spațiu.

**Restriții**

$2 \leq N \leq 5000$   
 $1 \leq M \leq 8000$

$P > 2$

Pentru datele de test, există întotdeauna soluție, nu neapărat unică.

**Exemplu**

jungla.in

5 6  
2 3  
3 5  
2 5  
2 1  
1 5  
4 5

jungla.out

4  
2 3 5 1

(Tabăra de pregătire a lotului național de informatică, Alba Iulia, 2004)

**Soluție**

Putem asocia problemei un graf neorientat (vârfurile grafului corespund celor  $N$  triburi; există muchie de la vârful  $i$  la vârful  $j$  dacă există drum direct între locațiile triburilor  $i$  și  $j$ ). Problema cere să se determine cel mai scurt ciclu elementar de lungime pară.

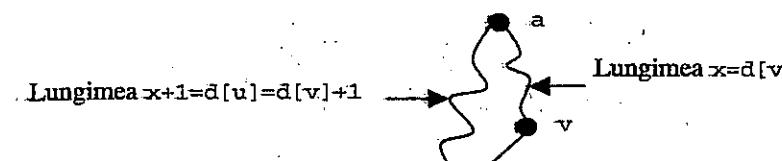
Vom efectua o parcurgere în lățime din fiecare vârf al grafului. De îndată ce este detectat un ciclu elementar de lungime pară, el va fi comparat cu cel mai scurt ciclu elementar de lungime pară și reținut, dacă este cazul.

Parcurgerea BFS va fi modificată astfel încât să detecteze ciclurile elementare de lungime pară. În acest scop, pentru fiecare vârf reținem următoarele informații:

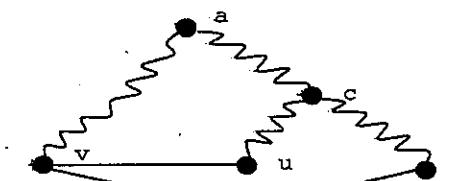
1.  $d[v]$ =distanța de la  $a$  (vârful de start al parcurgerii BFS) până la  $v$  (adică lungimea celui mai scurt lanț de la  $a$  la  $v$ ); dacă  $v$  nu a fost încă vizitat  $d[v]=+\infty$ ;
2.  $p[v]$ =vârful părinte al lui  $v$  în arborele BFS;  $p[v]=0$  dacă  $v=a$  sau  $v$  nu a fost încă vizitat; evident, dacă  $p[v] \neq 0$ ,  $d[v]=1+d[p[v]]$ ;
3.  $o[v]$ =perechea vârfului  $v$ ; dacă  $o[v] \neq 0$ , atunci  $o[v]$  este un vârf situat pe același nivel cu  $v$  astfel încât  $[v, o[v]]$  este muchie în graf. Dacă vârful  $v$  are pereche, atunci și  $o[v]$  are pereche și  $o[o[v]]=v$ ;
4.  $r[v]$ =cel mai îndepărtat strămoș al vârfului  $v$  în arborele BFS care are pereche; dacă  $v$  nu are un strămoș care are pereche, atunci  $r[v]=0$ .

În timpul parcurgerii BFS, atunci când vârful  $v$  este extras din coadă, urmează să se analizeze toate vârfurile adiacente cu  $v$ . Când este analizată muchia  $[v, u]$  în funcție de valorile  $d[u]$ ,  $o[v]$  și  $o[u]$  se procedează astfel:

1. dacă  $d[u]=d[v]-1$ , atunci această muchie a fost deja parcursă (în sens invers);
2. dacă  $d[u]=+\infty$ , atunci  $d[u]=d[v]+1$ ,  $p[u]=v$  și vârful  $u$  va fi plasat în coadă;
3. dacă  $d[u]=d[v]+1$ , un ciclu elementar de lungime pară a fost detectat; fie  $c$  cel mai apropiat strămoș comun al vârfurilor  $u$  și  $v$  în arborele BFS (să il notăm cu  $LCA(v, u)$ ), atunci lanțul de la  $c$  la  $v$  și lanțul de la  $c$  la  $u$  și muchia  $[v, u]$  constituie ciclul căutat;



4. dacă  $d[u]=d[v]$  și  $o[v]=u$ , această muchie a fost deja analizată în sens invers;
5. dacă  $d[u]=d[v]$  și  $o[v] \neq u$ , iar  $o[v] \neq 0$  și  $o[u] \neq 0$ , un ciclu elementar de lungime pară a fost detectat; fie  $x=o[v] \neq 0$  și  $c=LCA(x, u)$ , atunci lanțul de la  $c$  la  $x$ , lanțul de la  $c$  la  $u$  și muchiile  $[x, v]$  și  $[v, u]$  formează ciclul dorit;



6. dacă  $d[u]=d[v]$  și  $o[u]=o[v]=0$ , se verifică dacă  $r[u]=r[v]$ ; în acest caz,  $o[v]=u$ ,  $o[u]=v$ ; în caz contrar, a fost detectat un ciclu elementar de lungime pară; să presupunem că  $r[v]=x \neq 0$  și  $y=o[x]$ ; fie  $c=LCA(y, u)$ , atunci lanțul de la  $c$  la  $y$ , urmat de muchia  $[y, x]$ , apoi de lanțul de la  $x$  la  $v$ , urmat de muchia  $[v, u]$  și de lanțul de la  $u$  la  $v$  reprezintă ciclul dorit.

După ce sunt analizate toate vârfurile adiacente cu  $v$ , le vom parcurge din nou pentru a fixa  $r[u]$  pentru fiecare vârf  $u$  care a devenit fiu al lui  $v$  ( $r[u]=r[v]$ , exceptând cazul în care  $r[v]=0$  și  $o[v] \neq 0$  când  $r[u]=v$ ).

```
#include <stdio.h>
#include <stdlib.h>
#define NMax 10000
#define InFile "jungla.in"
#define OutFile "jungla.out"
#define Infinit NMax+10
typedef struct lista
{
    int nr; /*numarul de noduri din lista de adiacenta */
    int *l; /*adresa de inceput a listei de adiacenta */
} lista;
typedef lista graph[NMax];
int n, caz, bffound;
long int m;
graph G;
int d[NMax], p[NMax], o[NMax], r[NMax];
int C[NMax], B, E;
/*C este coada; B este inceputul, iar E sfarsitul cozii */
int Cycle[NMax], P;
/*ciclul curent; P este lungimea sa */
```

```

int MinCycle[NMax], Pmin=Infinit;
/*cel mai scurt ciclu; Pmin lungimea sa */
int changes=0, bf=0;

void read(void);
void init(int);
int BFS(int);
void write_solution(void);
void determine_cycle1(int, int);
void determine_cycle2(int, int);
void determine_cycle3(int, int);

int main(void)
{int a, found, i;
read();
for (a=1; a<=n; a++)
{found=BFS(a); bf++;
if (found)
if (P<Pmin)
{Pmin=P; changes++; bffound=bf-1; caz=found;
for (i=0; i<P; i++)
MinCycle[i]=Cycle[i];
}
if (Pmin==Infinit)
printf("Eroare! Nu exista solutie.\n");
write_solution();
return 0;
}

void read(void)
{FILE *in= fopen(InFile,"r");
long int i;
int x, y;
fscanf(in, "%d %d", &n, &m);
for (i=1; i<=n; i++) {G[i].l=NULL; G[i].nr=0;}
for (i=1; i<=m; i++)
{fscanf(in,"%d %d",&x,&y);
G[x].nr++;
G[x].l=realloc(G[x].l,G[x].nr*sizeof(int));
G[x].l[G[x].nr-1]=y;
G[y].nr++;
G[y].l=realloc(G[y].l,G[y].nr*sizeof(int));
G[y].l[G[y].nr-1]=x;
}
fclose(in); }

void init(int a)
{int i;
for (i=1; i<=n; i++)
{d[i]=Infinit;
p[i]=o[i]=r[i]=0;
d[a]=0; C[0]=a; B=E=0;
}
}

```

```

int BFS(int a)
{int i, v, u;
init(a);
while (B<=E)
{v=C[B++];
if (d[v]>Pmin+2) return 0;
for (i=0; i<G[v].nr; i++)
{u=G[v].l[i];
if (d[u]==d[v]-1) continue;
if (d[u]==Infinit)
{d[u]=i+d[v];
p[u]=v;
C[++E]=u;
continue; }
if (d[u]==d[v]+1)
{determine_cycle1(v,u);
return 1; }
if (d[u]==d[v] && o[v]==u) continue;
if (d[u]==d[v] && o[v]+o[u])
{determine_cycle2(v,u);
return 2; }
if (d[u]==d[v] && !o[u] && !o[v])
if (r[v]==r[u])
{o[u]=v; o[v]=u; }
else
{determine_cycle3(v,u);
return 3; }
}
for (i=0; i<G[v].nr; i++)
{u=G[v].l[i];
if (p[u]==v)
if (!r[v] && o[v])
r[u]=v;
else r[u]=r[v];
}
}
return 0;
}

void determine_cycle1(int v, int u)
{
int c1=v, c2=p[u], lg=1, i;
int P1[NMax/2], P2[NMax/2];
P1[0]=v; P2[0]=u; P2[1]=p[u];
while (c1!=c2)
{c1=p[c1];
P1[lg]=c1;
c2=p[c2];
P2[++lg]=c2; }
for (i=0; i<lg; i++)
{Cycle[i]=P1[lg-i-1];
Cycle[i+lg]=P2[i]; }
P=2*lg;
}

```

```

void determine_cycle2(int v, int u)
{
    int x, aux;
    int c1, c2, lg=1, i, j;
    int P1[NMax/2], P2[NMax/2];
    if (o[u]) {aux=u; u=v; v=aux;}
    x=o[v];
    c1=x; c2=u;
    P1[0]=x; P2[0]=u;
    while (c1!=c2)
        {c1=p[c1]; P1[lg]=c1;
         c2=p[c2]; P2[lg++]=c2;}
    Cycle[0]=v;
    for (i=1; i<=lg; i++) Cycle[i]=P1[i-1];
    for (j=lg-2; j>=0; j--, i++) Cycle[i]=P2[j];
    P=i;
}

void determine_cycle3(int v, int u)
{
    int c1, c2, lg1=1, lg2, i, lvx, aux, x, y, j, pozx, pozy;
    int xx, yy, dlca, lg;
    int P1[NMax/2], P2[NMax/2];
    if (r[u]) {aux=u; u=v; v=aux;}
    x=r[v]; y=o[x];
    P1[0]=y; P2[0]=u;
    c1=y; c2=u; lg1=lg2=1;
    while (p[c1]) {c1=p[c1]; P1[lg1++]=c1;}
    while (p[c2]) {c2=p[c2]; P2[lg2++]=c2;}
    dlca=-1;
    for (xx=0; xx<lg1; xx++)
        for (yy=0; yy<lg2; yy++)
            if (P1[xx]==P2[yy] && dlca<=d[P1[xx]])
                {dlca=d[P1[xx]]; pozx=xx; pozy=yy;}
    for (i=0; i<=pozx; i++) Cycle[i]=P1[pozx-i];
    lg=pozx+1;
    P1[0]=v; lvx=0;
    while (P1[lvx]!=x) {P1[lvx+1]=p[P1[lvx]]; lvx++;}
    for (j=lvx; j>=0; j--)
        Cycle[lg++]=P1[j];
    Cycle[lg++]=u;
    for (j=1; j<pozy; j++)
        Cycle[lg++]=P2[j];
    P=lg;
}

void write_solution()
{
    FILE *out=fopen(OutFile, "w");
    int i;
    fprintf(out, "%d\n", Pmin);
    for (i=0; i<Pmin-1; i++)
        fprintf(out, "%d ", MinCycle[i]);
    fprintf(out, "%d\n", MinCycle[Pmin-1]);
    fclose(out);
}

```

## 1.20. Aplicații propuse

### 1. Turn

Pe o masă se află N pahare de plastic, numerotate distinct de la 1 la N. Paharele pot fi puse unul în altul, formându-se astfel turnuri de pahare.

Se dă o secvență de mutări de forma a b, cu semnificația „turnul care conține paharul a va fi pus peste turnul care conține paharul cu numărul b”. Dacă paharele a și b sunt în același turn sau dacă a=b, atunci mutarea nu are nici un efect.

Mutările trebuie să fie executate în ordinea în care ele apar în secvență.

De exemplu, să considerăm că avem 7 pahare, iar sirul de mutări este 1 3, 2 6, 3 6, 4 7 și 4 2. Configurația paharelor de pe masă se schimbă astfel:

Inițial: (1) (2) (3) (4) (5) (6) (7)

După mutarea 1 3: (1 3) (2) (4) (5) (6) (7)

După mutarea 2 6: (1 3) (2 6) (4) (5) (7)

După mutarea 3 6: (1 3 2 6) (4) (5) (7)

După mutarea 4 7: (1 3 2 6) (4 7) (5)

După mutarea 4 2: (1 3 2 6 4 7) (5)

### Cerință

Scrieți un program care să determine mutarea 0, adică mutarea care ar trebui să fie executată înaintea primei mutări din secvență, astfel încât numărul de pahare din cel mai înalt turn să fie maxim.

### Date de intrare

Fișierul de intrare turn.in conține pe prima linie două numere naturale, N (numărul de pahare) și M (numărul de mutări), separate prin spațiu. Fiecare dintre următoarele M linii conține o mutare dată prin două numere naturale a b, separate printr-un spațiu.

### Date de ieșire

Fișierul de ieșire turn.out va conține o singură linie pe care se află mutarea 0, specificată ca o pereche de numere naturale a b, separate printr-un singur spațiu.

### Restricții

$2 \leq N \leq 10000$

$0 \leq M \leq 100000$

### Exemple

turn.in	turn.out	turn.in	turn.out
5 3	4 5	6 4	3 5
1 2		5 2	
5 3		3 3	
4 1		1 3	
		6 2	

(.campion, 2003)

## 2. Program

Atunci când este analizată o sursă scrisă într-un anumit limbaj de programare, este util să verificăm dacă există instrucțiuni care cu siguranță nu vor fi executate niciodată. Dacă există, atunci acestea ar putea reprezenta o eroare în program.

Să considerăm un limbaj de programare simplu, care admite următoarele instrucțiuni:

**EXECUTA** – execuția programului continuă cu următoarea instrucțiune;

**SALT n** – execuția programului continuă cu cea de-a n-a instrucțiune;

**SALT n SAU m** – execuția programului continuă cu cea de-a n-a sau cea de-a m-a instrucțiune.

Execuția unui program începe întotdeauna cu prima instrucțiune.

### Cerință

scrieți un program care să determine numărul de instrucțiuni ce nu vor fi executate niciodată.

### Date de intrare

Fișierul de intrare **program.in** conține mai multe instrucțiuni, câte o instrucțiune pe linie. Ultima linie din fișier conține ca marcaj de sfârșit caracterul ‘.’.

### Date de ieșire

Fișierul de ieșire **program.out** conține o singură linie pe care se află numărul de instrucțiuni care nu vor fi executate niciodată.

### Restricții

Numărul de instrucțiuni este cel mult egal cu 10000.

Instrucțiunile sunt numerotate începând cu 1.

### Exemple

program.in	program.out	program.in	program.out
SALT 1	1	EXECUTA	2
EXECUTA		SALT 4 SAU 6	
		EXECUTA	
		SALT 3	
		EXECUTA	
		SALT 8	
		EXECUTA	
		EXECUTA	

(.campion, 2003)

## 3. APM

Se consideră un arbore având costuri atașate pe muchii și un sir de numere naturale. Să se adauge arborelui un număr maxim de muchii având costuri dintre numerele date astfel încât graful obținut să aibă ca arbore parțial de cost minim arborele inițial.

### Date de intrare

Pe prima linie a fișierului **apm.in** se află numărul natural **n**, reprezentând numărul nodurilor din arbore. Pe următoarele **n-1** linii se găsesc câte trei numere

naturale separate prin căte un spațiu, sub forma  $i \ j \ c$ , având semnificația: există muchie de la  $i$  la  $j$  de cost  $c$ . Pe următoarele linii se află sirul de numere dat, căte un număr pe o linie.

### Date de ieșire

Pe fiecare linie a fișierului de ieșire **apm.out** se vor scrie căte trei numere  $i \ j \ c$ , separate prin căte un spațiu, având semnificația: „adăugăm o muchie între nodurile  $i$  și  $j$  având costul  $c$ ”.

### Restricții și precizări

$$0 < n < 256$$

Costurile muchiilor arborelui sunt numere naturale nenule  $\leq 32000$ .

În sirul dat există cel mult 32000 de numere naturale nenule  $\leq 30000$ .

Fiecare element din sir poate reprezenta costul unei singure muchii adăugate.

Elementele sirului nu sunt neapărat distinse.

Între oricare două noduri din graf va exista cel mult o muchie.

În cazul în care nu se poate adăuga nici o muchie, în fișierul de ieșire se va scrie o singura linie ce va conține valoarea 0.

În graful obținut după adăugarea muchiilor pot exister mai mulți arbori parțiali de cost minim, dar costul acestora trebuie să fie identic cu al arborelui inițial.

### Exemplu

apm.in	apm.out
4	1 4 5
1 2 2	2 3 3
3 4 4	
3 1 2	
2	
5	
3	

(.campion, 2004)

## 4. Zmeu

Zmeul le-a închis pe cele 5 fete ale împăratului în castelul său. Făt-Frumos a venit să le scoată din castel, folosind harta pe care i-a dat-o zâna bună. Pe hartă sunt desenate cele **n** camere ale castelului, numerotate de la 1 la **n**, și cele **m** tuneluri dintre camere. De asemenea, sunt marcate camerele în care se găsesc fetele și Făt-Frumos, timpul necesar traversării fiecarui tunel și cele **p** camere din care se poate ieși din castel.

### Cerință

Determinați timpul minim necesar lui Făt-Frumos pentru a lua cele 5 fete și a le scoate din castel.

### Date de intrare

Pe prima linie a fișierului de intrare **zmeu.in** sunt scrise numerele **n**, **m** și **p**, reprezentând numărul de camere, numărul de tuneluri și respectiv numărul de ieșiri din castel. Pe a doua linie sunt scrise numerele a 6 camere distincte: primul număr e camera în care se află Făt-Frumos, iar celelalte 5 sunt camerele fetelor. Pe următoarele **m** linii sunt căte 3 numere, reprezentând cele **m** tuneluri: primele două

reprezintă camerele între care face degătura tunelul, al treilea timpul necesar traversării tunelului. Ultimele  $p$  linii conțin câte un număr reprezentând camerele din care se poate ieși din castel. Toate numerele sunt întregi strict pozitive, iar numerele de pe aceeași linie sunt separate prin câte un spațiu.

#### Date de ieșire

Prima linie a fișierului zmeu.out va conține un singur număr reprezentând timpul minim necesar pentru a porni din camera lui Făt-Frumos, a trece prin camerele celor 5 fete și a ajunge într-o cameră care are ieșire în afara. Se va lua în considerare numai timpul traversării tunelurilor, neglijându-se timpul traversării camerelor sau a ieșirii din castel.

#### Restricții

$$4 \leq n \leq 1000$$

$$1 \leq p \leq n/2$$

Timpul de traversare a unui tunel este un număr întreg din intervalul [1, 200].

#### Exemplu

zmeu.in	zmeu.out
10 11 3	29
2 1 7 5 10 6	
1 7 3	
1 2 2	
1 3 4	
2 3 2	
2 8 1	
2 4 3	
3 4 2	
3 5 2	
3 6 5	
6 10 3	
6 9 4	
7	
4	
8	

(campion, 2005)

#### 5. Graf

Se știe că într-un graf neorientat conex, între oricare două vârfuri există cel puțin un lanț, iar lungimea unui lanț este egală cu numărul muchiilor care-l compun. Definim noțiunea *lanț optim între două vârfuri*  $x$  și  $y$  ca fiind un lanț de lungime minimă care are ca extremități vârfurile  $x$  și  $y$ . Este evident că între oricare două vârfuri ale unui graf conex vom avea unul sau mai multe lanțuri optime, depinzând de configurația grafului.

#### Cerință

Fiind dat un graf neorientat conex cu  $N$  vârfuri numerotate de la 1 la  $N$  și două vârfuri ale sale notate  $x$  și  $y$  ( $1 \leq x, y \leq N$ ,  $x \neq y$ ), se cere să scrieți un program care determină vârfurile care aparțin tuturor lanțurilor optime dintre  $x$  și  $y$ .

#### Date de intrare

Fișierul graf.in conține pe prima linie patru numere naturale reprezentând:  $N$  (numărul de vârfuri ale grafului),  $M$  (numărul de muchii),  $x$  și  $y$  (două vârfuri). Pe următoarele  $M$  linii se află câte două numere naturale nenule  $A_i$ ,  $B_i$  ( $1 \leq A_i, B_i \leq N$ ,  $A_i \neq B_i$ , pentru  $1 \leq i \leq M$ ) fiecare dintre aceste perechi de numere reprezentând căte o muchie din graf.

#### Date de ieșire

Fișierul graf.out va conține pe prima linie numărul de vârfuri comune tuturor lanțurilor optime dintre  $x$  și  $y$ ; pe a doua linie, numerele corespunzătoare etichetelor acestor vârfuri, dispuse în ordine crescătoare; între două numere consecutive de pe această linie se va afla câte un spațiu.

#### Restricții

$$2 \leq N \leq 7500$$

$$1 \leq M \leq 14000$$

#### Exemplu

graf.in	graf.out	graf.in	graf.out
6 7 1 4	3	3 2 1 3	2
1 2	1 4 5	1 2	1 3
1 3		3 1	
1 6			
2 5			
3 5			
5 6			
5 4			

(Olimpiada Județeană de Informatică, 2006)

#### 6. Trasee

Fiindcă nu m-am calificat la ONI, am vacanță de primăvară. Am luat harta rutieră a țării și am marcat  $n$  orașe pe care le consider interesante și merită vizitare. Am numerotat cele  $n$  orașe de la 1 la  $n$ . Orașul în care locuiesc este numerotat  $x$ .

Vacanța nu este lungă, așa că am hotărât că pot vizita exact  $k$  orașe situate pe un traseu care respectă simultan condițiile (numim un astfel de traseu *valid*):

- Traseul pornește din orașul  $x$  în care locuiesc (orașul  $x$ ).
- Între oricare două orașe consecutive de pe traseu trebuie să existe un drum direct (drum care nu trece prin alte orașe).
- Traseul trece prin exact  $k$  orașe distincte.
- Pentru orice oraș  $y$  de pe traseu, distanța (numărul de drumuri directe parcurse) pe traseul respectiv de la orașul  $x$  la orașul  $y$  (exprimată în număr de drumuri directe parcurse pe traseu) este minimă (adică nu există un alt traseu pe hartă de la  $x$  la  $y$  având lungime strict mai mică).

Două trasee valide sunt disjuncte dacă ele nu conțin un același drum direct.

#### Cerință

Cum eu nu m-am calificat la ONI, scrieți un program care să mă ajute să determin numărul maxim de trasee valide disjuncte.

**Date de intrare**

Fișierul de intrare `trasee.in` conține pe prima linie patru numere naturale  $n$ ,  $m$ ,  $x$  și  $k$  separate prin căte un spațiu, care reprezintă numărul de orașe, numărul de drumuri directe dintre orașe, numărul orașului în care locuiesc și respectiv numărul de orașe aflate pe un traseu. Următoarele  $m$  linii conțin cele  $m$  drumuri directe de pe hartă, căte un drum pe o linie. Fiecare drum direct este specificat prin două numere naturale distincte cuprinse între 1 și  $n$ , separate printr-un spațiu, reprezentând orașele conectate de drumul respectiv.

**Date de ieșire**

Fișierul de ieșire `trasee.out` va conține o singură linie pe care va fi scris numărul maxim de trasee valide disjuncte.

**Restricții și precizări**

$$1 \leq n \leq 200; 1 \leq x, k \leq n$$

Între două orașe poate exista cel mult un drum direct bidirecțional.

**Exemplu**

<code>trasee.in</code>	<code>trasee.out</code>
7 8 1 3	3
1 3	
2 3	
5 3	
4 6	
4 7	
1 4	
6 1	
6 7	

**Explicație**

Orașul de plecare este orașul 1. Există 4 trasee valide care trec prin exact 3 orașe, plecând din 1 (1-3-2, 1-3-5, 1-4-7, 1-6-7). Numărul maxim de trasee valide disjuncte este 3. O soluție ar putea fi 1-3-2, 1-4-7, 1-6-7.

(Olimpiada Națională de Informatică, Tîrgoviște, 2006)

**7. Grade**

Fie  $n$  un număr natural nenul și un sir de  $n$  numere naturale  $d_1, d_2, \dots, d_n$ .

**Cerință**

Scrieți un program care să determine un graf conex care are secvența gradelor vârfurilor  $d_1, d_2, \dots, d_n$ .

**Date de intrare**

În fișierul de intrare `grade.in` se află pe prima linie un număr natural  $n$ , iar pe linia a doua cele  $n$  valori naturale separate prin spații.

**Date de ieșire**

Fișierul de ieșire `grade.out` va conține pe fiecare linie căte două numere naturale (cuprinse între 1 și  $n$ ), separate printr-un spațiu  $x$   $y$ , cu semnificația „în graful conex obținut există muchie între vârful  $x$  și vârful  $y$ ”.

**Restricții**

$$1 \leq n \leq 5000$$

Vârfurile grafului vor fi numerotate de la 1 la  $n$ .

Nu este necesar ca vârful 1 să aibă gradul  $d_1$ , vârful 2 să aibă gradul  $d_2$  etc. Două secvențe de grade sunt considerate egale dacă după sortare ele coincid.

**Exemple**

<code>grade.in</code>	<code>grade.out</code>
5	5 1
2 1 1 2 4	4 1
	3 2
	3 1
	1 2

(Tabăra de pregătire a lotului național de informatică, Alba Iulia, 2004)

**8. Traseu**

Gigel tocmai a fost admis la facultate și acum studiază oferta de cursuri a facultății. În ofertă facultății există  $N$  cursuri (pe care Gigel le-a numerotat de la 1 la  $N$ ). Fiecare curs are alocat un anumit număr de credite (să notăm cu  $c_i$  numărul de credite alocate cursului  $i$ , pentru orice  $i=1, 2, \dots, N$ ).

Evident, cursurile nu sunt total independente. De exemplu, pentru a studia cursul de *Fundamente algebrice ale informaticii* sunt necesare noțiuni din cursul *Algebra*. Mai exact, spunem că un curs  $j$  depinde de cursul  $i$  dacă în cursul  $j$  apar noțiuni care au fost definite în cursul  $i$ .

Relațiile de dependență sunt definite corect (adică nu există situații în care anumite cursuri să nu poată fi studiate; de exemplu, cursul  $i$  să depindă de cursul  $j$ , iar cursul  $j$  să depindă de cursul  $i$ ).

Dacă Gigel optează să studieze un curs  $i$ , el trebuie să studieze în prealabil toate cursurile de care depinde cursul  $i$ .

În plus, Gigel observă că toate cursurile sunt grupate pe niveluri de complexitate. De exemplu, toate cursurile de inițiere (cursuri în care nu intervin noțiuni dintr-un alt curs) constituie nivelul de complexitate 0. Pe nivelul de complexitate 1 sunt toate cursurile în care intervin noțiuni din cursurile de pe nivelul de complexitate 0. și aşa mai departe, pe nivelul de complexitate  $x$  se află toate cursurile în care intervin noțiuni din cursuri situate pe nivelurile de complexitate 0, 1, ...,  $x-1$ .

Gigel dorește să își construiască un traseu universitar. Traseul universitar preferat de Gigel va fi format dintr-o succesiune de cursuri distincte  $T=(t_1, t_2, \dots, t_k)$  care respectă următoarele condiții:

1. dacă în traseul  $T$  apare cursul  $t_i$ , atunci toate cursurile de care depinde cursul  $t_i$  apar în  $T$  înaintea cursului  $t_i$ ;
2. traseul  $T$  conține cel puțin un curs de pe fiecare nivel de complexitate;
3. suma creditelor alocate cursurilor de pe traseul  $T$  este egală cu  $M$ ;
4. numărul de cursuri studiate să fie minim.

**Cerință**

Scrieți un program care să determine un traseu universitar care să respecte condițiile din enunț.

**Date de intrare**

Fiecare fișier de test conține:

- pe prima linie, numerele naturale  $N$  și  $M$  separate printr-un spațiu, reprezentând numărul de cursuri și respectiv numărul total de credite;

- pe cea de-a doua linie, N numere naturale separate prin câte un spațiu  $c_1 \ c_2 \ ... \ c_N$ , reprezentând numărul de credite alocate fiecăruia curs;
- pe fiecare dintre următoarele linii, câte două numere naturale separate printr-un spațiu  $i \ j$  ( $1 \leq i \leq N$ ,  $1 \leq j \leq N$ ,  $i \neq j$ ) cu semnificația „cursul  $j$  depinde de cursul  $i$ ”.

#### Date de ieșire

Pentru fiecare fișier de intrare, veți genera un fișier de ieșire denumit x-traseu.out (unde x va reprezenta numărul testului corespunzător). Fiecare fișier de ieșire va conține pe prima linie un număr natural k reprezentând numărul de cursuri de pe traseul universitar al lui Gigel (minim posibil). Pe cea de-a doua linie vor fi scrise k numere naturale distincte, separate prin câte un spațiu, reprezentând, în ordine, cursurile de pe traseul universitar al lui Gigel.

#### Exemplu

traseu.in	traseu.out
12 27	6
2 10 3 6 1 4 3 8 5 1 1 2	12 8 7 10 2 3
1 5	
3 11	
6 11	
5 6	
4 5	
12 8	
8 7	
8 9	
7 10	

(Tabăra de pregătire a lotului național, Sibiu, 2005)

#### 9. Loc

Elevii de la Liceul de Informatică vor să își aleagă un loc de întâlnire. Ei au luat harta orașului Iași și au marcat punctele culturale interesante (Casa Pogor, Bolta Rece etc.), apoi le-au numerotat de la 1 la N. Liceul de Informatică este numerotat cu 0. Între oricare două puncte culturale există cel mult o stradă care poate fi parcursă în ambele sensuri.

Pentru a avea cât mai multe variante de a evita întâlniri nedorite, elevii au hotărât să aleagă drept loc de întâlnire un punct pentru care există un număr maxim de drumuri de lungime minimă care pleacă de la liceu.

#### Cerință

Scrieți un program care să aleagă un loc de întâlnire pentru elevii de la Liceul de Informatică.

#### Date de intrare

Fișierul de intrare loc.in conține pe prima linie numărul natural N, reprezentând numărul de puncte culturale interesante. Pe cea de a doua linie se află un număr natural M reprezentând numărul de străzi. Pe următoarele M linii sunt descrise străzile, câte o stradă pe o linie. Pe linia  $i+2$  se află 3 numere naturale  $x_i \ y_i \ L_i$ , cu semnificația „există o stradă între punctele  $x_i$  și  $y_i$  de lungime  $L_i$ ”.

#### Date de ieșire

Fișierul de ieșire loc.out conține pe prima linie un număr natural P, reprezentând locul de întâlnire ales. Pe cea de-a doua linie este scris numărul de drumuri de lungime minimă de la Liceul de Informatică până la locul de întâlnire. Pe cea de-a treia linie este scrisă distanța de la liceu până la locul de întâlnire.

#### Restricții

$$N \leq 100$$

$$x_i, y_i \in \{0, 1, \dots, N\}, \forall i \in \{1, 2, \dots, m\}$$

$$0 < L_i < 10000, \forall i \in \{1, 2, \dots, m\}$$

#### Exemplu

loc.in	loc.out
4	4
6	3
0 1 10	40
0 4 40	
1 2 10	
2 3 10	
1 4 30	
2 4 20	

(Concursul „Urmașii lui Moisil”, 2002)

#### 10. Trip

Gigel s-a hotărât să facă o călătorie prin țară. El vrea să meargă până într-un oraș destinație, iar apoi să se întoarcă în orașul din care a plecat. Pentru ca excursia să nu devină monotonă, Gigel a pus următoarea condiție: drumul de la dus și cel de la întoarcere trebuie să conțină un număr minim de șosele comune. În plus, nici drumul de la dus, nici cel de la întoarcere nu au voie să parcurgă aceeași șosea de mai multe ori.

#### Cerință

Scrieți un program care să-l ajute pe Gigel să găsească un drum dus-întors care să respecte condițiile specificate.

#### Date de intrare

Pe prima linie a fișierului trip.in se află două numere întregi S și D ( $S \neq D$ ) care reprezintă orașul de plecare și orașul destinație. Următoarea linie conține două numere întregi N și M, unde N este numărul de orașe și M este numărul de șosele. Următoarele M linii conțin câte două numere întregi P și Q ( $1 \leq P, Q \leq N$ ,  $P \neq Q$ ) cu semnificația că între orașele P și Q există o șosea (pe care se poate merge în ambele sensuri). Orașele sunt numerotate de la 1 la N.

#### Date de ieșire

Prima linie a fișierului trip.out trebuie să conțină un număr întreg care reprezintă numărul minim de șosele comune pe care le au cele două drumuri. A doua linie trebuie să conțină o succesiune de orașe care reprezintă drumul de la S la D (inclusiv S și D). A treia linie trebuie să conțină un drum de întoarcere de la D la S. Dacă nu există două astfel de drumuri, fișierul de ieșire va conține -1.

**Restricții**  
 $3 \leq N \leq 1000$   
 $2 \leq M \leq 100000$

**Exemple**

trip.in  
 1 6  
 7 8  
 2 1  
 1 3  
 2 3  
 4 2  
 4 5  
 5 6  
 7 5  
 6 7

trip.out  
 2  
 1 3 2 4 5 7 6  
 6 5 4 2 1

(campion, 2003)

### 11. Turism

În orașul Adelton din insula Zanzibar există o agenție de turism. Agenția a decis să ofere clienților ei vizite la diferite obiective turistice. Pentru a câștiga cât mai mulți bani, agenția a luat următoarea decizie: este necesară găsirea unui drum de lungime minimă care începe și se termină în același loc.

În oraș sunt  $N$  intersecții numerotate de la 1 la  $N$  și  $M$  străzi bidirectionale. Două intersecții pot fi unite prin mai multe străzi, dar nu există nici o stradă care are ambele extremități în aceeași intersecție. Orice traseu este o secvență de străzi  $y_1, \dots, y_k$ ,  $k > 2$ . Strada  $y_i$  ( $1 \leq i \leq k$ ) leagă intersecțiile  $x_i$  și  $x_{i+1}$ , iar strada  $y_k$  conectează intersecțiile  $x_k$  și  $x_1$ . Toate numerele de ordine ale intersecțiilor prin care se trece ( $x_1, \dots, x_k$ ) sunt diferite două câte două. Lungimea unui traseu este egală cu suma lungimilor străzilor din care este alcătuit traseul.

**Cerință**

scrieți un program care găsește un traseu de lungime minimă. În cazul în care nu există nici un traseu, programul va afișa 0.

**Date de intrare**

Pe prima linie a fișierului turism.in se află două numere întregi  $N$  și  $M$  reprezentând numărul de intersecții și respectiv de străzi. Pe următoarele  $M$  linii se află descrierile străzilor, câte o stradă pe o linie. Fiecare linie conține 3 numere întregi: cele două intersecții unite de stradă și lungimea străzii corespunzătoare liniei.

**Date de ieșire**

Fișierul turism.out va conține o singură linie pe care va fi scris fie 0, fie traseul de lungime minimă determinat. Traseul este descris prin intersecțiile sale, separate prin câte un spațiu.

**Restricții și precizări**

$N \leq 100$   
 $M \leq 10000$

Lungimile străzilor sunt numere naturale mai mici decât 500.

**Exemplu**

traseu.in	traseu.out
5 7	1 3 5 2
1 4 1	
1 3 300	
3 1 10	
1 2 16	
2 3 100	
2 5 15	
5 3 20	

(campion, 2003)

### 12. Lanț de alarmă

Elevii unei școli au decis să formeze un lanț de alarmă. Fiecare dintre ei își alege un unic succesor, căruia urmează să îi transmită toate mesajele primite.

Un lanț de alarmă funcționează astfel: cineva trimite un mesaj unui elev. Acest elev va transmite mesajul succesorului său și aşa mai departe până când mesajul ajunge din nou la elevul care l-a primit inițial.

**Cerință**

Scrieți un program care să determine numărul minim de modificări ce trebuie să fie efectuate pentru a transforma configurația inițială în lanț de alarmă.

**Date de intrare**

Fișierul de intrare alarma.in conține pe prima linie numărul de elevi  $N$  ( $1 < N < 256$ ). Fiecare dintre cele  $N$  linii care urmează în fișier conține două nume separate prin semnul >. O linie de forma A>B are semnificația că B este succesorul lui A. Numele au cel mult 20 de caractere.

**Date de ieșire**

Fișierul de ieșire alarma.out va conține o singură linie pe care va fi scris numărul minim de modificări necesare pentru a transforma configurația din fișierul de intrare în lanț de alarmă.

**Exemplu**

alarm.in	alarm.out
10	4
Anita>Peter	
Andrew>Julia	
David>Andrew	
Natalie>Gabriella	
Edith>David	
Peter>Anita	
Gabriella>Julius	
Adam>David	
Julia>Gabriella	
Julius>Julia	

(Olimpiada de Informatică a Europei Centrale, 2005)

### 13. Ziduri

Un ziar are redacția la etajul unui clădiri. Acest etaj de formă pătratică este alcătuit numai din camere de aceeași dimensiune și de formă pătratică. Pentru un etaj cu  $4 \times 4$  camere avem configurația :



Unele dintre zidurile camerelor lipsesc. Directorul și redactorul-șef au biroul în camere separate. Directorul are biroul în camera de pe linia 1 și coloana 1, iar redactorul-șef în camera de pe ultima linie și ultima coloană. Deplasarea între două camere vecine se poate face numai dacă ele nu au zid despărțitor. Pentru a mări viteza de deplasare între birourile directorului și redactorului-șef se ia decizia că unele ziduri să fie desființate. Un studiu făcut de departamentul administrativ arată că deplasarea între două camere fără zid conduce la un cost de o unitate monetară, iar deplasarea între două camere care au avut zid și a fost dărâmat conduce la un cost de p unități monetare.

#### Cerință

Determinați costul minim al unei deplasări de la camera directorului la camera redactorului-șef. Dintre toate deplasările de cost minim, determinați numărul minim de ziduri ce trebuie dărâmate într-o astfel de deplasare.

#### Date de intrare

Fișierul de intrare `ziduri.in` conține pe prima linie  $n$  (numărul de camere de pe o linie, respectiv coloană) și  $p$ , costul trecerii de la o cameră la alta între care s-a dărâmat zidul despărțitor, cele două numere fiind separate printr-un spațiu. Pe următoarele  $n$  linii se află câte  $n$  numere naturale din mulțimea  $\{0, 1, \dots, 15\}$  separate prin câte un spațiu. Aceste numere naturale transformate în baza 2 (pe 4 biți) ne dau informații despre existența zidurilor în jurul camerei (1 pentru zid și 0 în caz contrar). De exemplu, dacă un astfel de număr are reprezentarea  $abcd$  în baza 2, atunci a este pentru zidul dinspre vest, b pentru cel din nord, c pentru cel din est, iar d pentru cel din sud.

#### Date de ieșire

Fișierul de ieșire `ziduri.out` va conține pe prima linie costul minim al deplasării de la director la redactorul-șef, iar pe linia a doua numărul minim de ziduri dărâmate.

#### Restricții și precizări

$1 < n < 101$   
 $0 < p < 101$

### Exemplu

`ziduri.in`  
4 3  
9 1 1 0  
12 5 5 1  
1 5 5 4  
4 6 12 0

`ziduri.out`  
.8  
.1

(Olimpiada Națională de Informatică, Galați, 2005)

### 14. Team

Sâmbăta o petrec cu prietenii la discoteca „Why not“ din centrul orașului. De acolo plecăm în zori  $p$  persoane, în gașcă, și trebuie să ajungem fiecare acasă. Atât la discotecă, la punctele de destinație ale membrilor găștii, precum și în alte puncte ale orașului se află stații de Team-Taxi pe care le putem folosi în drumul spre casă, plătind frățește pe fiecare segment de drum o sumă fixă pe care o pretinde șoferul mașinii (în funcție de lungimea drumului și nu în funcție de numărul de pasageri).

În orice stație pot părăsi gașca numai cei care au ajuns la destinație, în acest caz *grupurile omogene* formate urmând să se despartă (pentru că au dispărut oamenii de legătură) și să ia în continuare taxiuri cu diferite destinații. Două grupuri omogene diferite pot merge în aceeași destinație, dar utilizând taxiuri diferite. Numim grup omogen o formăție de persoane cu numere de ordine consecutive. De exemplu, pentru  $p=6$ , de la discoteca pornește gașca în formația 1 2 3 4 5 6. Dacă la o stație se oprește persoana numărul 3, atunci se formează două *grupuri omogene*, 1 2 și 4 5 6. El se despart luând două taxiuri diferite. Două grupe formate din 1 4 5 și 2 6 **nu sunt omogene**.

Atâtă timp cât  $k$  persoane merg cu un taxi pe un segment pe care orice șofer cere invariabil suma  $c$ , contribuția fiecărui pe acel segment este  $c/k$ . Dacă o persoană merge singură cu un taxi pe segmentul respectiv, ea va trebui să plătească singură întreaga sumă.

#### Cerință

Știind numărul de persoane, rețeaua de stații de taxiuri, costurile de transport pe fiecare segment al rețelei și punctele de destinație ale fiecărui membru al găștii, se cere să se determine costul minim pe care îl pot plăti în total membrii găștii, astfel încât, utilizând taxiurile în maniera descrisă, fiecare să ajungă acasă.

#### Date de intrare

Din fișierul `team.in` se citesc :

- $p$  numărul de persoane din gașcă ce pleacă de la discoteca, de pe linia 1 ;
- $n$  numărul de stații de taxi, de pe linia 2 ; stațiile sunt numerotate de la 1 la  $n$  ;
- $m$  numărul de segmente ce leagă direct căte două stații, de pe linia 3 ;
- $m$  triplete de forma  $i \ j \ c$  ( $i$  și  $j$  sunt stații între care se consideră segmentul, iar  $c$  costul de transport pe segmentul respectiv), de pe următoarele  $m$  linii ;
- $d_1 \ d_2 \dots d_p$  stațiiile de destinație ale membrilor găștii (nu neapărat distinse).

**Date de ieșire**

Fișierul team.out va conține o singură linie cu un număr natural reprezentând costul minim determinat.

**Restricții și precizări**

$$1 \leq p \leq 50$$

$$2 \leq n \leq 500$$

$$0 \leq c \leq 1000$$

Se consideră stația 1 ca punct de plecare (discoteca).

Po toate străzile se poate circula în ambele sensuri.

**Exemplu**

team.in	team.out	Explicație
4	6	Grup Traseu Cost
5		1 2 3 4 1 → 3 4
8		1 2 3 4 3 → 2 1
1 2 6		persoana 2 rămâne la destinație (stația 2)
1 3 4		1 2 → 5 0
3 4 8		persoana 1 rămâne la destinație (stația 5)
2 4 1		3 4 2 → 4 1
3 5 7		persoanele 3, 4 rămân la destinație (stația 4)
2 3 1		Cost total 4+1+0+1=6
1 5 6		
2 5 0		
5 2 4 4		

(Tabăra de pregătire a lotului de informatică, Ploiești, 2006)

## 2. Liste înlățuită

Întâlnim și utilizăm liste în fiecare zi: la școală, profesorul are câte o listă cu elevii fiecărei clase – catalogul; la restaurant găsim o listă cu produsele oferite și prețurile lor – meniu; la o firmă de calculatoare vom primi o listă de componente și prețuri – oferta ș.a.m.d. Prin urmare, utilizăm liste ori de câte ori este necesar să organizăm într-o formă secvențială un ansamblu de informații.

*Listă* este o structură de date abstractă constituită dintr-o succesiune de elemente. Fiecare element din listă are un succesor (cu excepția ultimului element al listei) și un predecesor (cu excepția primului element din listă).

Cel mai simplu mod (dar nu întotdeauna și cel mai eficient) este de a memoria elementele listei într-un vector. Succesiunea elementelor este în acest caz implicită (ordinea indicilor vectorului).

În cazul în care structura de date este dinamică (suportă frecvent operații de inserare, respectiv ștergere a unor elemente din structură) această implementare a listelor nu este eficientă. O altă implementare posibilă, eficientă pentru structuri de date liniare dinamice, sunt listele înlățuite. În funcție de numărul de legături existente între elemente consecutive ale listei, lista este simplu sau dublu înlățuită.

### 2.1. Liste simplu înlățuite

O *listă simplu înlățuită* este o structură de date constituită dintr-o succesiune de elemente denumite noduri. Fiecare nod din listă conține două părți: o parte de informație (în care sunt memorate informațiile corespunzătoare nodului, specifice problemei) și o parte de legătură (în care este memorată adresa următorului element din listă).

Pentru simplitate vom considera, fără a restrângere generalitatea, că informațiile memorate în nodurile listei sunt numere întregi. Declarația structurii care reprezintă un nod din listă va fi:

```
struct Nod
{
    int inf;
    struct Nod * urm;
};
```

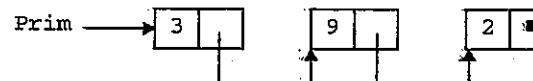
Observăm că adresa fiecărui nod din listă este conținută într-un alt nod, cu o singură excepție: primul nod al listei. Prin urmare, pentru a identifica o listă simplu înălțuită este suficient să reținem adresa primului element al listei.

Vom defini tipul de date **Lista**, care este adresa unui nod din listă:

```
typedef struct Nod * Lista;
```

Exceptând primul nod (a cărui adresă nu este conținută de nici un alt nod al listei), într-o listă simplu înălțuită mai există un nod special: ultimul nod. După acest nod nu mai urmează nici un alt nod al listei, deci ultimul nod va conține în partea de legătură pointerul **NULL**.

Putem figura o listă simplu înălțuită cu elemente întregi astfel:



Am figurat prin săgeți legăturile dintre noduri. **Prim** este un pointer care reține adresa primului element din listă. Simbolul **■** semnifică marcajul de sfârșit de listă (pointerul **NULL**).

Operațiile fundamentale pe care trebuie să le implementăm sunt:

1. Inserarea unui nod în listă;
2. Stergerea unui nod din listă;
3. Consultarea unei liste.

### Inserarea unui nod într-o listă simplu înălțuită

Pentru inserare vom implementa o funcție denumită **Insert** cu trei parametri:

1. **Prim** – pointer care conține adresa primului nod din lista în care se face inserarea; acest parametru va fi transmis prin referință, deoarece în urma inserării începutul listei se poate modifica;
2. **p** – pointer care conține adresa nodului din listă după care se face inserarea; dacă **p** este **NULL**, deducem că inserarea noului nod se va face la începutul listei;
3. **x** – informația nodului care urmează să fie inserat în listă.

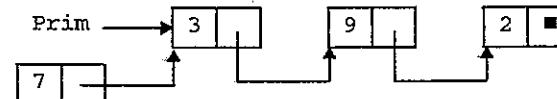
Vom aloca dinamic memorie pentru un nou nod, a cărui adresă o vom reține în variabila pointer **q**. În zona de informații vom memora valoarea **x** (**q->inf=x**).

La inserare apar două cazuri distincte. Dacă **p==NULL** noul nod va fi inserat la începutul listei; dacă **p!=NULL** inserarea se face în interiorul listei.

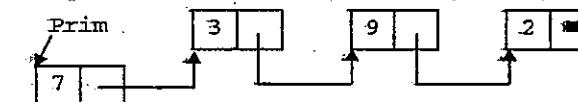
#### Exemplu

Să ilustrăm pas cu pas inserarea la începutul listei din exemplul precedent a unui nou nod, cu informația 7, a cărui adresă va fi memorată în pointerul **q**:

1. Creăm o legătură de la noul nod către primul nod din listă, memorând în câmpul **urm** al noului nod adresa primului nod din listă (**q->urm=Prim**).



2. În acest moment primul nod din listă este noul nod (**Prim=q**).

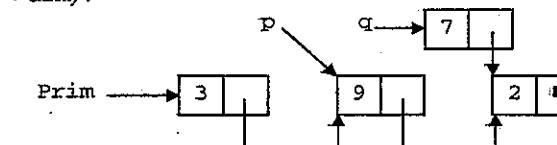


În cel de-al doilea caz trebuie să inserăm noul nod după nodul a cărui adresă este memorată în parametrul **p**.

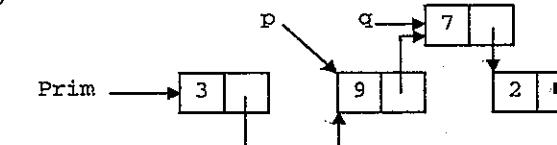
#### Exemplu

Să ilustrăm pas cu pas inserarea în lista din exemplul precedent a unui nou nod cu informația 7, după nodul cu informația 9.

1. Creăm o legătură de la noul nod către nodul care urmează după nodul indicat de **p** (**q->urm=p->urm**):



2. Modificăm legătura de la nodul indicat de **p**; după acest nod va urma noul nod (**p->urm=q**).



```

void Insert(Lista & Prim, Lista p, int x)
{
    Lista q=new Nod;
    q->inf=x;
    if (!p) { q->urm=Prim; Prim=q; }
    else { q->urm=p->urm; p->urm=q; }
}
    
```

#### Observații

1. Crearea unei liste se realizează prin inserări succesive de elemente.
2. Funcția de inserare creată este generală, funcționează pentru toate cazurile posibile (inserare la începutul listei, inserare în interiorul listei și inserare la sfârșitul listei).

### Stergerea unui nod dintr-o listă simplu înălțuită

Pentru stergere vom implementa o funcție denumită **Delete** cu doi parametri:

1. **Prim** – pointer care conține adresa primului nod al listei din care se face stergere; acest parametru va fi transmis prin referință, deoarece în urma stergerii, începutul listei se poate modifica;
2. **p** – pointer care conține adresa nodului din listă care precedă nodul ce urmează a fi sters (se transmite adresa nodului precedent și nu adresa nodului de sters pentru

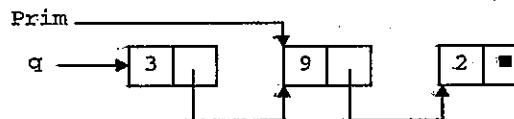
în putea restaura corect legăturile în listă; dacă  $p$  este NULL deducem că va fi șters primul nod din listă.

Și la ștergere apar două cazuri distincte. Dacă  $p==NULL$  va fi șters primul nod din listă; dacă  $p!=NULL$  va fi șters un nod din interiorul listei.

#### *Exemplu*

Să ilustrăm pas cu pas ștergerea primului nod al listei din exemplul precedent:

- Memorăm în variabila  $q$  adresa nodului ce urmează a fi șters ( $q=Prim$ ), apoi adresa primului element din listă se schimbă ( $Prim=Prim->urm$ ).



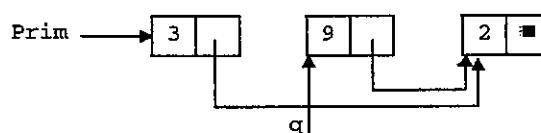
- La sfârșit eliberăm zona de memorie alocată nodului ce a fost eliminat din listă (delete  $q$ ).

În cel de-al doilea caz trebuie să ștergem nodul care urmează după nodul a căruia adresă este memorată în parametrul  $p$ .

#### *Exemplu*

Să ilustrăm pas cu pas ștergerea din lista din exemplul precedent a nodului cu informația 9.

- Memorăm în variabila  $q$  adresa nodului ce urmează să fie șters ( $q=p->urm$ ), apoi modificăm legătura care există de la  $p$  la  $q$  și o transformăm într-o legătură la nodul care urmează după cel ce va fi șters ( $p->urm=q->urm$ ).



- La sfârșit eliberăm zona de memorie alocată nodului ce a fost eliminat din listă (delete  $q$ ).

```
void Delete(Lista & Prim, Lista p)
(Lista q;
if (p)
  (q=p->urm;
   if (q) (p->urm=q->urm; delete q; )
  )
else
  (q=Prim;
   if (q) (Prim=Prim->urm; delete q; )
  )
```

#### *Parcursarea unei liste*

A parcurge o listă presupune a vizita fiecare nod din listă, în scopul prelucrării informațiilor asociate nodurilor.

În acest scop vom utiliza un pointer  $p$  căruia îi vom atribui inițial adresa primului element din listă. Cât timp lista nu s-a terminat ( $p!=NULL$ ), se vizitează nodul curent, apoi se trece la nodul următor ( $p=p->urm$ ). Funcția următoare realizează parcursarea unei liste simplu înlănuite cu afișarea informațiilor din noduri :

```
void Afisare(Lista Prim)
(Lista p;
for (p=Prim; p; p=p->urm)
  cout<<p->inf<<' ';
cout<<endl;
)
```

#### *Crearea unei stive*

Așa cum știm, *stiva* este o structură de date abstractă care suportă două operații: inserarea unui nod la începutul listei și ștergerea primului nod din listă.

Stiva poate fi implementată cu ajutorul unei liste simplu înlănuite. Crearea unei stive cu informații citite de la tastatură se realizează astfel :

```
Prim=NULL;
cin>>n;
for (i=1; i<=n; i++)
  {cin>>x;
   Insert(Prim, NULL, x); }
```

#### *Observație*

Valorile sunt plasate în stivă în ordinea inversă a introducerii lor.

#### *Crearea unei cozi*

*Coadă* este o structură de date abstractă care suportă două operații: inserarea unui element după ultimul nod din coadă și ștergerea primului element din coadă.

Pentru a realiza eficient cele două operații este bine să reținem pe lângă adresa primului nod și adresa ultimului nod din coadă (în variabila *Ultim*).

Crearea unei cozi cu elemente citite de la tastatură se realizează astfel :

```
Prim=Ultim=NULL;
cin>>n;
for (i=1; i<=n; i++)
  {cin>>x;
   if (Prim) //coada nu este vida
     { Insert(Prim, Ultim, x);
       Ultim=Ultim->urm; }
   else //coada este vida
     { Insert(Prim, NULL, x);
       Ultim=Prim; }
```

*Observație*

Valorile sunt plasate în coadă în ordinea introducerii lor.

*Crearea unei liste ordonate*

Pentru a crea o listă ordonată va trebui ca înainte de a insera o nouă valoare în listă să căutăm poziția sa corectă, apoi să inserăm valoarea pe poziția corectă, astfel încât la fiecare moment lista să fie sortată.

Dacă lista este vidă sau valoarea care trebuie să fie inserată este mai mică decât informația primului nod din listă, atunci inserarea se realizează la începutul listei.

Altfel, vom lua un pointer *p* pe care îl deplasăm de la începutul listei până când nodul care urmează după cel indicat de *p* are o informație mai mare decât valoarea de inserat sau *p* este ultimul nod din listă. Inserarea noii valori se va realiza după nodul indicat de *p*.

```
Prim=NULL;
cin>>n;
for (i=1; i<=n; i++)
{
    cin>>x;
    if (!Prim || Prim->inf>x)
        Insert(Prim, NULL, x);
    else
    {
        for (p=Prim; p->urm && p->urm->inf<x; p=p->urm);
        Insert(Prim, p, x);
    }
}
```

*Interclasarea a două liste ordonate*

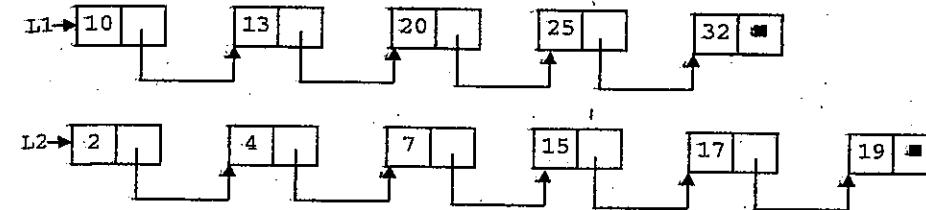
Să considerăm două liste simplu înălțuite ordonate crescător cu adresele de început *L1* și respectiv *L2*. Intenționăm să obținem o a treia listă, cu adresa de început *L3*, care să conțină informațiile din ambele liste în ordine crescătoare.

O primă variantă de rezolvare a problemei este de a construi o nouă listă, copiind succesiv elementele. Se parcurg simultan cele două liste cu ajutorul a doi pointeri. La fiecare pas se compară elementele curente și se copiază în lista rezultat elementul cu informația mai mică, trecând mai departe doar în lista din care s-a făcut copierea. Când una dintre liste s-a terminat, se copiază în lista rezultat nodurile rămase în cealaltă listă.

O altă abordare (mai interesantă) este de a contopi cele două liste, prin modificarea legăturilor dintre noduri astfel încât să se obțină o singură listă cu informațiile ordonate crescător. Evident, în acest mod cele două liste inițiale se distrug.

*Exemplu*

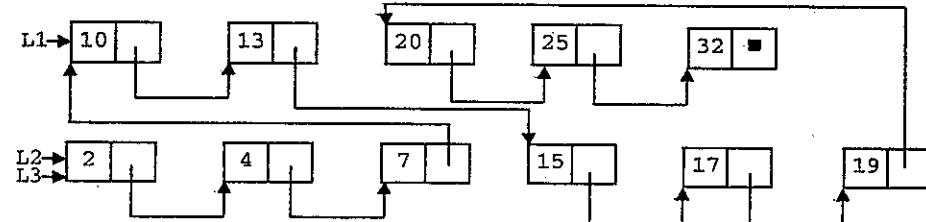
Să considerăm listele din figura următoare:



Deoarece informația din primul nod al celei de-a două liste este mai mică decât informația din primul nod al primei liste, deducem că adresa de început a listei rezultat va fi adresa primului nod din cea de-a două listă.

Observăm că primele 3 elemente din lista a doua au informațiile mai mici decât primul element din prima listă, deci, în lista rezultat, după al treilea element din cea de-a două listă trebuie să urmeze primul element din prima listă. Primele două elemente din prima listă au informațiile mai mici decât al patrulea element din a două listă. Prin urmare, după al doilea element din prima listă trebuie să urmeze al patrulea element din a două listă. În sfârșit, ultimele 3 elemente din cea de-a două listă au informațiile mai mici decât al treilea element din prima listă, deci după ultimul element din cea de-a două listă va urma al treilea element din prima listă.

Se obține lista :



*Lista Interclasare(Lista L1, Lista L2)*

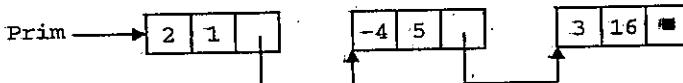
```

if (!L1) return L2;
if (!L2) return L1;
/* L3 indica începutul listei rezultat
   p indica nodul curent din lista în care ne deplasam
   q indica nodul curent din cealaltă listă */
L3=L2; p=L2; q=L1;
if (L1->inf<L2->inf) {L3=L1; p=L1; q=L2;}
while (q)
    {
        while (p->urm && p->urm->inf<=q->inf) p=p->urm;
        aux=p->urm;
        p->urm=q;
        p=q;
        q=aux; }
return L3;
}
```

### Polinoame în formă condensată

Un polinom rar (în care majoritatea coeficienților sunt nuli) poate fi reprezentat în formă condensată ca o listă simplu înlățuită în care reținem monoamele în ordinea crescătoare a gradelor. Pentru fiecare monom vom reține în partea de informații a nodului corespunzător coeficientul și gradul.

De exemplu, polinomul rar  $P(X)=3X^{16}-4X^5+2X$  poate fi reprezentat în formă condensată astfel:



Vom modifica declarația structurii care reprezintă un nod din listă astfel:

```
struct Nod
{
    int c, g;
    struct Nod * urm; };
}
```

### Evaluarea unui polinom memorat în formă condensată

Pentru a evalua un polinom în formă condensată, vom parcurge lista, pentru fiecare nod din listă adăugând la valoarea curentă a polinomului valoarea monomului curent.

```
int eval (Lista Prim, int x0)
{
    int val=0, gc=0, p=1;
    Lista q;
    for (q=Prim; q; q=q->urm)
        //calculez puterea
        for (; gc<q->g; p*=x0, gc++);
        val+=p*q->c;
    }
    return val;
}
```

### Suma a două polinoame memorate în formă condensată

Vom parcurge simultan listele corespunzătoare celor două polinoame. La fiecare pas se inserează în lista corespunzătoare polinomului sumă monomul cu gradul cel mai mic. În cazul în care monoamele curente au același grad, în lista suma se inserează un monom care are drept coeficient suma coeficienților celor două monoame curente (dacă această sumă este nenulă).

```
List Suma(Lista L1, Lista L2)
{
    Lista L3=NULL, Ultim=NULL, p=L1, q=L2;
    while (p && q)
        if (p->g<q->g)
            {Insert(L3, Ultim, p->c, p->g);
             p=p->urm;
             if (Ultim) Ultim=Ultim->urm; else Ultim=L3; }
        else
            if (p->g>q->g)
                {Insert(L3, Ultim, q->c, q->g);
                 q=q->urm;
                 if (Ultim) Ultim=Ultim->urm; else Ultim=L3; }
```

```

    else
        if (p->c+q->c)
            {Insert(L3, Ultim, p->c+q->c, p->g);
             if (Ultim) Ultim=Ultim->urm; else Ultim=L3; }
        p=p->urm; q=q->urm; }

    while (p)
        {Insert(L3, Ultim, p->c, p->g);
         p=p->urm; Ultim=Ultim->urm; }

    while (q)
        {Insert(L3, Ultim, q->c, q->g);
         q=q->urm; Ultim=Ultim->urm; }

    return L3;
}
```

### Exerciții propuse

1. Scrieți o expresie care să aibă valoarea 1 dacă și numai dacă nodurile ale căror adrese sunt memorate în pointerii *p* și respectiv *q* sunt consecutive în listă.
2. Variabila *L* reține adresa primului element dintr-o listă simplu înlățuită de 10 elemente. Care este adresa celui de-al patrulea element din listă?
3. Variabila *L* reține adresa primului element al unei liste simplu înlățuite. Scrieți o expresie cu valoarea 1 dacă și numai dacă lista are exact două noduri.
4. Scrieți o funcție care să primească ca parametru adresa primului nod al unei liste simplu înlățuite și care să returneze numărul de noduri ale listei.
5. Un element aflat la adresa *q* face parte din lista *L* dacă la sfârșitul executării secvenței următoare, variabila *r* are valoarea :

```
r=p;
while (r!=q && r) r=r->adr;
```

a. *p*                  b. *q*                  c. *NULL*

d. 1

(Bacalaureat, august 2003)

6. Într-o listă simplu înlățuită cu cel puțin patru elemente, fiecare element reține în câmpul *adr* adresa elementului următor din listă. Dacă *p* și *q* sunt adresele a două noduri din listă astfel încât *p==q->adr*, atunci nodul de la adresa *q* se află în listă :

a. pe o poziție neînvecinată cu <i>p</i>	b. imediat înaintea nodului de la adresa <i>p</i>
c. imediat după nodul de la	d. în aceeași poziție ca și nodul de la
adresa <i>p</i>	adresa <i>p</i>

(Bacalaureat, iulie 2006)

7. Scrieți o funcție care să primească drept parametru adresa primului element al unei liste simplu înlățuite și care să returneze adresa ultimului element al listei.
8. Scrieți o funcție care să parcurgă o listă simplu înlățuită a cărei adresă de început este specificată ca parametru și care să returneze numărul de perechi de noduri consecutive în listă, care conțin în partea de informație valori întregi de semne contrare.

9. Scrieți o funcție cu un singur parametru (adresa primului nod dintr-o listă simplu înănguită) care să șteargă din lista specificată toate nodurile care au ca informație valoarea 0.
10. Scrieți o funcție cu un singur parametru (adresa primului nod dintr-o listă simplu înănguită) care să returneze adresa unui nod situat la mijlocul listei. Funcția va parcurge lista o singură dată.
11. Scrieți o funcție care să afișeze, în ordinea inversă memorării lor, informațiile din nodurile unei liste simplu înănguțite primele ca parametru. Funcția nu va utiliza structuri de date suplimentare.
12. Scrieți o funcție care să inverseze legăturile într-o listă simplu înănguită specificată ca parametru. Funcția va returna adresa primului nod din lista obținută după inversarea legăturilor.
13. Variabilele p și q memorează adresele de început ale listelor liniare nevide simplu înănguțite L<sub>1</sub>, respectiv L<sub>2</sub>. Elementele listelor sunt de același tip, iar în câmpul adr este reținută adresa elementului următor. Variabilele p, q, r și u sunt de același tip.
- a. Pentru a determina numărul de elemente ale listei L<sub>1</sub> se utilizează o variabilă întreagă n astfel :

a. n=0; r=p;  
while(r) {r=r->adr; n++;}

c. n=0; r=p;  
while(r->adr) {r=r->adr; n++;}

b. n=0; r=p;  
while(r) r=r->adr; n++;

d. n=0; r=p;  
do (n++) while (r);

b. Funcția egale(ad1,ad2) returnează valoarea 1 dacă informațiile memorate la adresele ad1 și ad2 coincid, altfel returnează 0. Secvența următoare calculează în variabila întreagă n numărul de elemente din lista L<sub>1</sub>:

n=0; r=p;  
while (egale(r,p) && r)  
{r=r->adr; n++; }

a. care sunt egale două câte două  
c. egale consecutive aflate la  
începutul listei

b. care sunt egale cu primul element  
d. distințe consecutive aflate la  
începutul listei

c. Trebuie să fie mutat primul element al listei L<sub>1</sub> imediat după primul element al listei L<sub>2</sub>, în rest listele rămânând neschimbate. Care dintre atribuirile următoare sunt necesare și în ce ordine se efectuează?

a. r=q->adr;      b. r=p->adr;      c. q->adr=p;  
d. p->adr=r;      e. p=r;      f. p->adr=q->adr;

b f e c | b f c e | b c f e | a f c e  
d. Pentru a uni listele L<sub>1</sub> și L<sub>2</sub>, plasând lista L<sub>1</sub> în continuarea listei L<sub>2</sub> se efectuează operațiile:

a. r=q; while(r->adr) {r=r->adr; r->adr=p;}      b. r=q; while(r->adr) r=r->adr;  
c. r=p; while(r) r=r->adr;      d. r=p; while(r->adr) r=r->adr;  
r->adr=q;

- e. Dacă L<sub>1</sub> este organizată ca stivă, stabiliți care este operația corectă de extragere din stivă:
- a. r=p->adr; p=r; delete p;      b. r=p->adr; p=r; delete r;  
c. r=p; p=p->adr; delete r;      d. delete p; p=p->adr;
- f. Dacă L<sub>1</sub> este o coadă, cu p adresa primului și u adresa ultimului element, iar r este adresa unui element ce urmează a fi adăugat în coadă (r->adr fiind NULL), stabiliți care dintre următoarele este o operație corectă de adăugare:
- a. r->adr=p; p=r;      b. r=u->adr; u=r;      c. u->adr=r; u=r;

(Bacalaureat, iulie 2003)

14. Se consideră 10 liste, orice element de listă având un câmp adr ce memorează adresa elementului următor. Știind că p este un vector de adrese, p[i] reprezentând adresa de început a listei L<sub>i</sub>, decideți care dintre următoarele secvențe de instrucții afișează numerele de ordine ale listelor nevide.

a.   
for (i=0; i<10; i++)  
if (!p[i]->adr) cout<<i;  
c.   
for (i=0; i<10; i++)  
if (p[i]->adr) cout<<i;  
d.   
for (i=0; i<10; i++)  
if (p[i]) cout<<i;

(Simulare Bacalaureat, 2003)

15. Care este efectul următoarei secvențe de instrucții?

Lista L, p, q;  
L=new Nod; L->inf=1;  
p=new Nod; p->inf=2; L->urm=p;  
q=new Nod; q->inf=3; p->urm=q;  
for (p=L; p; p=p->urm) cout<<p->inf;

- a. Va afișa pe ecran 123      b. Va afișa pe ecran 321      c. Va afișa pe ecran 312  
d. Va genera eroare la execuție      e. Va genera eroare la compilare

#### 16. Matrice rare

O matrice este considerată rară dacă cel puțin 70% dintre elementele sale sunt nule. Pentru a memora eficient o matrice rară putem utiliza o listă simplu înănguită în care pentru fiecare element nenul al matricii vom reține linia și coloana pe care se află elementul, precum și valoarea sa. Considerăm că elementele matricii sunt memorate în listă în ordinea liniilor, iar în cazul în care există mai multe elemente nenele pe o același linie, acestea sunt memorate în listă în ordinea coloanelor.

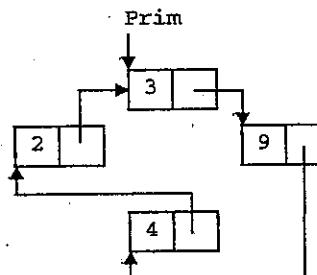
- a. Scrieți o funcție care să primească drept parametru adresa primului nod al unei liste simplu înănguțite ce memorează în formă condensată o matrice rară și care să afișeze matricea în fișierul rar.out în formă uzuală (pe linii, elementele de pe același linie fiind separate prin spații).  
b. Scrieți o funcție care să primească drept parametru adresa primului nod dintr-o listă simplu înănguită care memorează în formă condensată o matrice rară și care returnează suma elementelor de pe diagonala principală a matricii.

- c. Scrieți o funcție care să primească drept parametru adresa primului nod dintr-o listă simplu înlățuită ce memorează în formă condensată o matrice rare și care să determine numărul maxim de elemente nerule situate pe aceeași linie.
- d. Scrieți o funcție care să primească ca parametri adresele de început ale două liste simplu înlățuite reprezentând două matrice rare cu același număr de linii și același număr de coloane și care să construiască o a treia listă simplu înlățuită reprezentând forma condensată a sumei celor două matrice. Funcția va returna adresa de început a listei rezultat.

## 2.2. Liste simplu înlățuite circulare

O listă simplu înlățuită este *circulară* dacă după ultimul element din listă urmează primul element al listei.

*Exemplu*



*Observații*

- Nici unul dintre nodurile unei liste circulare nu conține valoarea NULL în partea de legătură. Operațiile elementare asupra listelor simplu înlățuite circulare vor fi implementate înțând cont de această observație.
- Pentru a identifica o listă simplu înlățuită circulară putem reține adresa oricărui element din listă.

### Inserarea unui nod într-o listă simplu înlățuită circulară

Pentru inserare vom implementa o funcție denumită *Insert* cu trei parametri:

- Prim – pointer care conține adresa primului nod din lista în care se face inserarea; acest parametru va fi transmis prin referință, deoarece în urma inserării începutul listei se poate modifica;
- p – pointer care conține adresa nodului din listă după care se face inserarea;
- x – informația nodului care urmează să fie inserat în listă.

Vom aloca dinamic memorie pentru un nou nod, a cărui adresă o vom reține în variabila pointer q. În zona de informații vom memora valoarea x (q->inf=x).

La inserare apar două cazuri distincte. Dacă *Prim==NULL* (lista este vidă) noul nod inserat va fi singurul element din listă (deci în partea de legătură vom reține adresa sa); dacă *Prim!=NULL* inserarea se face în interiorul listei.

```

void Insert(Lista & Prim, Lista p, int x)
{
    Lista q=new Nod;
    q->inf=x;
    if (!Prim) {q->urm=q; Prim=q; }
    else {q->urm=p->urm; p->urm=q; }
}
  
```

### Stergerea unui nod dintr-o listă simplu înlățuită circulară

Pentru stergere vom implementa o funcție denumită *Delete* cu doi parametri:

- Prim – pointer care conține adresa primului nod al listei din care se face stergerea; acest parametru va fi transmis prin referință, deoarece în urma stergerii, începutul listei se poate modifica;
- p – pointer care conține adresa nodului din listă care precedă nodul ce urmează a fi sters (se transmite adresa nodului precedent și nu adresa nodului de sters pentru a putea restaura corect legăturile în listă).

Și la stergere apar două cazuri distincte. Dacă *p->urm==Prim* va fi sters primul nod din listă, deci valoarea parametrului *Prim* va fi actualizată (va primi adresa următorului nod din listă sau NULL dacă lista era formată dintr-un singur nod); dacă *p->urm!=Prim* va fi sters un nod oarecare din interiorul listei.

```

void Delete(Lista & Prim, Lista p)
{
    Lista q=p->urm;
    p->urm=q->urm;
    if (q==Prim)
        {Prim=Prim->urm;
         if (Prim->urm==Prim)
             Prim=NULL; }
    delete q;
}
  
```

### Parcursarea unei liste simplu înlățuite circulare

Pentru a parcurge o listă simplu înlățuită circulară, vom utiliza un pointer p, căruia îi vom atribui inițial adresa primului element din listă. La fiecare pas, se vizitează nodul curent, apoi se trece la nodul următor (*p=p->urm*), până când se revine la nodul de plecare. Funcția următoare realizează parcursarea unei liste simplu înlățuite circulare cu afișarea informațiilor din noduri:

```

void Afisare(Lista Prim)
{
    Lista p=Prim;
    if (!Prim) return;
    do
        {cout<<p->inf<< ' '; p=p->urm; }
    while (p!=Prim);
    cout<<endl;
}
  
```

### *Exerciții propuse*

1. Se consideră următoarea secvență de apeluri ale funcției de inserare a unei valori într-o listă simplu înlăntuită circulară (variabila  $L$  fiind inițial NULL):  
 $\text{Insert}(L, L, 1); \text{Insert}(L, L, 2); \text{Insert}(L, L \rightarrow \text{urm}, 3);$   
 $\text{Insert}(L, L, 4); \text{Insert}(L, L \rightarrow \text{urm} \rightarrow \text{urm}, 5);$   
 Ce se va obține prin apelarea funcției Afisare( $L$ )?
  2. Ce valoare returnează următoarea funcție, al cărei parametru reprezintă adresa unui nod dintr-o listă simplu înlăntuită circulară:

```

int f(Lista p)
{int v=0;
 Lista q;
 for (q=p; q->urm!=p; q=q->urm) v++;
 return v; }

```

3. Scrieți o funcție care să primească ca parametru adresa unui nod dintr-o listă simplu înlăncuită circulară nevidă și care să returneze adresa nodului ce precedă în listă nodul a cărui adresă este specificată ca parametru.

4. Într-o listă circulară simplu înlăncuită cu cel puțin două elemente, fiecare element memorarează în câmpul `adr` adresa elementului următor din listă. Știind că adresele `p` și `q` reprezintă adresele a două elemente distincte din listă, atunci elementul memorat la adresa `p` este succesorul elementului memorat la adresa `q` în listă dacă și numai dacă :

a. <code>p-&gt;adr==q</code> c. <code>p-&gt;adr-&gt;adr==q</code>	b. <code>q-&gt;adr==p</code> d. <code>q-&gt;adr-&gt;adr==p</code>
--	--

(Bacalaureat spécial, 2004)

5. Variabilele  $p$  și  $q$  conțin adresele a două noduri consecutive dintr-o listă simplu înăncălită circulară, iar variabila  $r$  conține adresa unui nod ce trebuie să fie inserat între nodurile indicate de  $p$  și respectiv  $q$ . În ce ordine trebuie să fie executate următoarele atribuiri pentru a realiza inserarea?

$q=p; r->urm=p; q->urm=r; p=p->urm;$

6. Din fișierul `poligon.in` se citește de pe prima linie numărul de vârfuri ale unui poligon convex, apoi se citesc coordonatele vârfurilor poligonului, date în ordină trigonometrică. Să se construiască o listă simplu înăncălită circulară, care să memoreze în ordinea dată vârfurile poligonului, apoi să se determine aria.

#### **7. Jocul lui Joseph**

Un grup format din  $n$  copii, numerotăți de la 1 la  $n$ , joacă următorul joc.

- se aşează în cerc, în ordinea 1, 2, ..., n, după copilul n urmând copilul 1;
  - copilul 1 alege un număr p și, începând cu el, copiii numără de la 1 la p;
  - copilul care a rostit numărul p este eliminat din joc, iar numărătoarea se reia după copilul care urmează celui eliminat.

Simulați acest joc utilizând o listă simplu înlăncuită circulară. Numerele naturale nenule  $n$  și  $p$  se vor citi de la tastatură. Pe ecran se va afișa ordinea în care sunt eliminate copiii din joc.

### **2.3. Liste dublu înlăntuite**

O listă dublu înlanțuită este o structură de date, constituită dintr-o succesiune de elemente denumite noduri. Fiecare nod din listă conține trei părți: o parte de informație (în care sunt memorate informațiile corespunzătoare nodului, specifice problemei) și două părți de legătură (în care este memorată adresa următorului element din listă și respectiv adresa precedentului element din listă).

Pentru simplitate vom considera, fără a restrânge generalitatea, că informațiile memorate în nodurile listei sunt numere întregi. Declarația structurii care reprezintă un nod dintr-o listă dublu înăntuită cu informații întregi va fi :

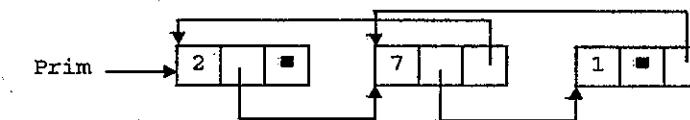
```

struct Nod
    {int inf;
     struct Nod * urm, * pre; }
typedef struct Nod * Lista;

```

### *Exempli*

Putem figura o listă dublu înlăntuită cu elemente întregi astfel



*Inserarea unui nod într-o listă dublu înlățuită*

Pentru inserare vom implementa o functie denumita `Insert` cu trei parametri

1. Prim – pointer care conține adresa primului nod din lista în care se face inserarea ; acest parametru va fi transmis prin referință, deoarece în urma inserării începutul listei se poate modifica ;
  2. p – pointer care conține adresa nodului din listă după care se face inserarea ; dacă p este NULL deducem că inserarea noului nod se va face la începutul listei ;
  3. x – informația nodului care urmează să fie inserat în listă.

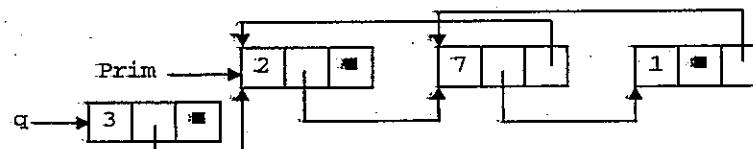
Vom aloca dinamic memorie pentru un nou nod, a cărui adresă o vom reține în variabila pointer  $q$ . În zona de informații vom memora valoarea  $x$  ( $q->inf=x$ ).

La inserare apar două cazuri distincte. Dacă  $p==NULL$  noul nod va fi inserat la începutul listei; dacă  $p!=NULL$  inserarea se face în interiorul listei.

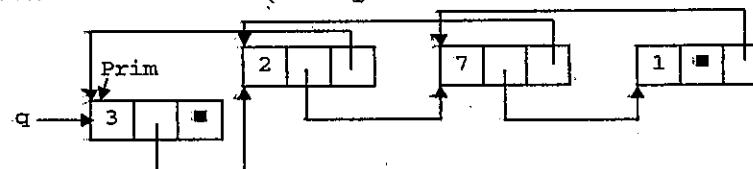
### *Exemplu*

Să ilustrăm pas cu pas inserarea la începutul listei din exemplul precedent a unui nou nod, cu informația 3, nodul cărui adresă va fi memorată în pointerul *q*:

1. Creăm o legătură de la nouul nod către primul nod din listă, memorând în câmpul `urm` al nouului nod adresa primului nod din listă (`q->urm=Prim`). Nouul nod va fi primul nod din listă, deci nu are precedent (`q->pre=NULL`).



2. Precedentul primului nod din listă va fi  $q$  ( $Prim \rightarrow pre = q$ ), iar primul nod din listă este de acum noul nod ( $Prim = q$ ).

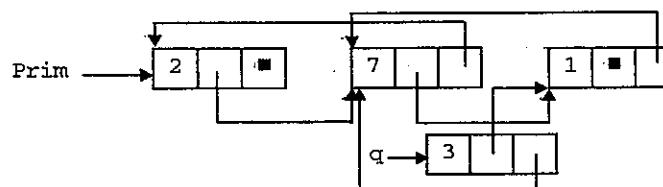


În cel de-al doilea caz trebuie să inserăm noul nod după nodul a cărui adresă este memorată în parametrul  $p$ .

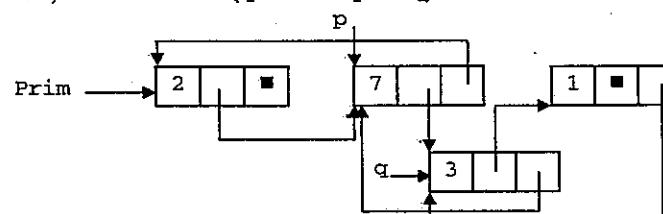
#### Exemplu

Să ilustrăm pas cu pas inserarea în lista din exemplul precedent a unui nou nod cu informația 3, după nodul cu informația 7.

1. Creăm legăturile dintre noul nod și precedentul său în listă ( $q \rightarrow pre = p$ ), respectiv succesorul său în listă – nodul indicat de  $p \rightarrow urm$  ( $q \rightarrow urm = p \rightarrow urm$ ):



2. Modificăm legăturile dintre nodurile din listă: după nodul indicat de  $p$  va urma noul nod ( $p \rightarrow urm = q$ ), iar înaintea nodului care urma după cel indicat de  $p$  (dacă acesta există) va fi noul nod ( $q \rightarrow urm \rightarrow pre = q$ ).



```
void Insert(Lista & Prim, Lista p, int x)
(Lista q=new Nod;
 q->inf=x; q->pre=p;
 if (!p) //inserare la inceputul listei
 {q->urm=Prim;
 if (Prim) Prim->pre=q;
 Prim=q; }
```

```
else
{q->urm=p->urm; p->urm=q;
if (q->urm) q->urm->pre=q;
}
```

#### Stergerea unui nod dintr-o listă dublu înlănuțuită

Pentru stergere vom implementa o funcție denumită `Delete` cu doi parametri:

1.  $Prim$  – pointer care conține adresa primului nod al listei din care se face stergerea; acest parametru va fi transmis prin referință, deoarece în urma stergerii, începutul listei se poate modifica;
2.  $q$  – pointer care conține adresa nodului din listă ce urmează a fi sters.  
Pentru simplitate, vom utiliza doi pointeri suplimentari –  $p$  care reține adresa nodului ce precedă nodul de sters și  $r$  care reține adresa nodului ce succede nodul de sters. Dacă  $p == NULL$  va fi sters primul nod din listă și va trebui să actualizăm valoarea parametrului  $Prim$ .

```
void Delete(Lista & Prim, Lista q)
(Lista p=q->pre, r=q->urm;
if (p) p->urm=r;
else Prim=r;
if (r) r->pre=p;
delete q; }
```

#### Consultarea unei liste dublu înlănuțuite

Consultarea unei liste dublu înlănuțuite se realizează în mod similar cu cea a unei liste simplu înlănuțuite, numai că o listă dublu înlănuțită poate fi consultată atât în sens direct (utilizând legăturile  $urm$ ), cât și în sens invers (utilizând legăturile  $pre$ ).

#### Exerciții propuse

1. Scrieți o funcție care să primească ca parametru adresa unui nod dintr-o listă dublu înlănuțită și care să returneze valoarea 1 dacă lista are proprietatea palindromică și 0 în caz contrar. Spunem că lista are proprietatea palindromică dacă, parcurgând lista de la început către sfârșit, obținem aceeași secvență de informații ca la parcurgerea listei de la sfârșit către început.
2. Într-o listă dublu înlănuțită cu cel puțin 4 elemente, fiecare element reține în câmpurile  $adp$  și  $adu$  adresa elementului precedent și respectiv adresa elementului următor din listă. Dacă  $p$  reprezintă adresa primului element din listă, iar  $q$  este de același tip cu  $p$ , atunci secvența următoare realizează:

```
q=p->adu->adp;
p=q->adu; p->adp=NULL;
delete q;
```

- a. interschimbarea primelor două elemente
- b. eliminarea primului element
- c. eliminarea celui de-al doilea element
- d. eliminarea ultimului element

(Bacalaureat, 2005)

3. Se introduc într-o listă dublu înlățuită, inițial vidă, valorile 1, 2, 3, 4, în această ordine. Care este secvența apelurilor funcției de inserare pentru a obține în listă informațiile în ordinea 3 1 4 2?
4. Cum trebuie să fie apelată funcția de stergere astfel încât să fie șters penultimul element al unei liste dublu înlățuite cu cel puțin 5 elemente, știind că adresa primului nod este L, iar adresa ultimului nod este SL?
5. Scrieți o funcție care să primească drept parametru adresa primului nod al unei liste dublu înlățuite cu număr impar de noduri și care să șteargă elementul situat la mijlocul listei.
6. O listă dublu înlățuită circulară este o listă dublu înlățuită în care după ultimul element al listei urmează primul element. Implementați cele trei operații elementare (inserare, stergere, consultare) pentru liste dublu înlățuite circulare.
7. Care dintre următoarele funcții realizează stergerea unui nod indicat de parametrul q din lista dublu înlățuită cu adresa de început L?

a.  
**void Sterge(Lista &L, Lista &q)**  
~~(if (q->pre) q->urm->pre=q->urm;~~  
~~if (q->urm) q->pre->urm=q->pre;~~  
~~delete q;~~

c.  
**void Sterge(Lista &L, Lista &q)**  
~~(if (q->pre) q->pre->urm=q->urm;~~  
~~else L=q->urm;~~  
~~if (q->urm) q->urm->pre=q->pre;~~  
~~delete q;~~

b.  
**void Sterge(Lista &L, Lista &q)**  
~~(Lista p=q; q=q->pre;~~  
~~q->urm=q->urm->urm;~~  
~~q->urm->urm->pre=q;~~  
~~delete p; )~~

d.  
**void Sterge(Lista &L, Lista &q)**  
~~(Lista p=q; q=q->urm;~~  
~~q->pre=q->pre->pre;~~  
~~q->pre->pre->urm=q;~~  
~~delete p; )~~

### 3. Structuri de date arborescente

Pentru problemele în care se impune o ierarhizare a informațiilor, astfel încât anumite elemente să fie subordonate altora, este utilă introducerea unei noi structuri de date – arbore cu rădăcină.

Arborii cu rădăcină sunt o prezență cotidiană. Fiecare dintre noi și-a reconstituit probabil la un moment dat arborele genealogic și, după cum vom vedea, cea mai mare parte a termenilor folosiți în limbajul informatic derivă de aici.

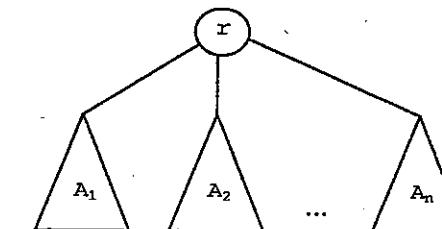
#### 3.1. Terminologie

Un *arbore cu rădăcină* este o structură de date constituită dintr-o mulțime finită de elemente denumite noduri, mulțime care satisfac una dintre următoarele două condiții :

1. este vidă ;
2. conține un nod special denumit rădăcina arborelui, toate celelalte noduri fiind particionate în  $n \geq 0$  arbori cu rădăcină  $A_1, A_2, \dots, A_n$ .

Arborii  $A_1, A_2, \dots, A_n$  sunt denumiți *subarbori* ai nodului rădăcină, iar nodurile conținute în subarbori sunt considerate *descendenți* ai nodului rădăcină. Rădăcinile arborilor  $A_1, A_2, \dots, A_n$  sunt denumite *descendenți direcți* sau *fiu* ai rădăcinii arborelui. Rădăcina arborelui este considerată nod *părinte* (sau *tată*) al rădăcinilor arborilor  $A_1, A_2, \dots, A_n$ .

Pentru a ilustra vizual relația de tip tată-fiu, rădăcina arborelui ( $r$ ) este unită printr-un segment de fiecare dintre rădăcinile subarborilor săi.



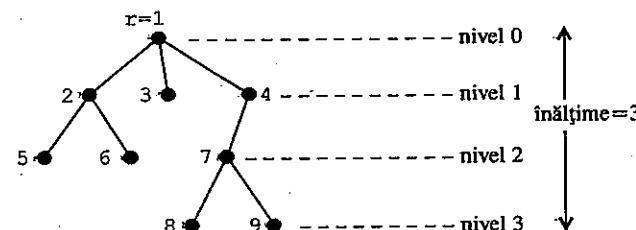
Există un drum unic de la rădăcina arborelui la fiecare nod  $x$  din arbore. Dacă  $y$  se găsește pe drumul unic de la rădăcină la  $x$ , atunci  $y$  se numește **ascendent** (strâmoș) al lui  $x$ .

Dacă un nod nu are descendenți el se numește nod **terminal** sau **frunză**. Două noduri care au același părinte se numesc **frați**.

#### *Observații*

O astfel de structură poate fi considerată un graf orientat special, care nu conține circuite, fiecare vîrf (exceptând vîrful rădăcină, care are gradul interior 0) fiind extremitatea finală a unui singur arc. Să observăm că alegând într-un mod arbitrar un vîrf drept rădăcină, orice graf neorientat conex și fără cicluri (adică un arbore în sensul utilizat în capitolul „Teoria grafurilor”) este un arbore cu rădăcină în sensul definiției de mai sus.

Din definiție putem deduce ierarhizarea nodurilor arborelui. Rădăcina arborelui constituie nivelul 0 în arbore. Rădăcinile subarborelor (adică fiile rădăcinii) constituie nivelul 1. Fiile fiilor rădăcinii vor constitui nivelul 2 și.a.m.d., fiile nodurilor de pe nivelul  $i$  vor constitui nivelul  $i+1$ .

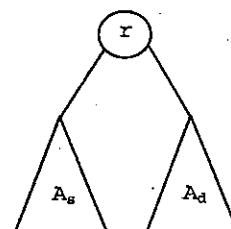


Nivelul maxim din arbore se numește **înălțimea (adâncimea)** arborelui.

În exemplul de mai sus, nodul 4 este un ascendent al lui 8. Nodurile 5, 6, 3, 8, 9 sunt noduri terminale. Nodurile 8, 9 sunt frați, iar descendenții nodului 4 sunt nodurile 7, 8, 9.

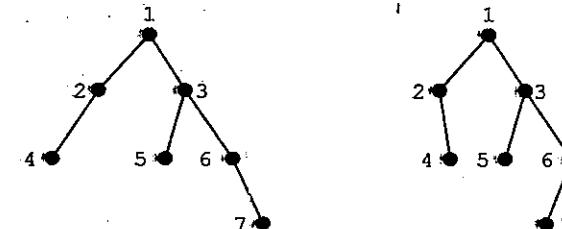
## 3.2. Arbori binari

O clasă foarte importantă de arbori cu rădăcină o constituie arborii binari. Un arbore binar este un arbore cu rădăcină în care fiecare nod are cel mult doi subarbore, denumiți subarborele stâng și subarborele drept. Rădăcina subarborelui stâng se numește fiu stâng, iar rădăcina subarborelui drept se numește fiu drept al rădăcinii.



#### *Exemple*

În figurile următoare sunt ilustrați doi arbori binari distincți cu 7 noduri. Ambii arbori au rădăcina 1 și înălțimea 3. În primul arbore, nodul 4 este fiu stâng al lui 2, iar 7 este fiu drept al lui 6. În cel de-al doilea, 4 este fiu drept al lui 2, iar 7 este fiu stâng al lui 6.



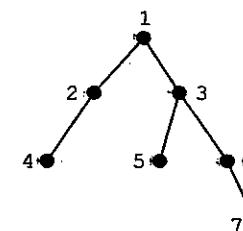
#### *Tipuri speciale de arbori binari*

##### *Arbore binar strict*

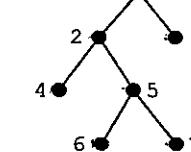
Un arbore binar se numește **strict** dacă orice nod din arbore are exact doi fiu sau este terminal.

##### *Exemplu*

Arbore binar care nu este strict:



Arbore binar strict:

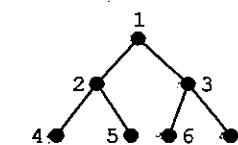


##### *Arbore binar plin*

Un arbore binar se numește **plin** dacă are  $2^k - 1$  noduri dispuse pe nivelurile 0, 1, ...,  $k-1$ , astfel încât pe fiecare nivel  $i$  se găsesc  $2^i$  vîrfuri.

##### *Exemplu*

Arborele binar plin cu înălțimea 2 este:



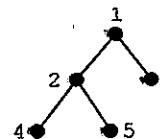
**Arbore binar complet**

Se numește **arbore binar complet** un arbore binar obținut dintr-un arbore binar plin prin eliminarea eventuală din dreapta către stânga a unor noduri de pe ultimul nivel. Mai exact, pentru a construi un arbore binar complet cu  $n$  noduri determinăm numărul natural  $k$  astfel încât  $2^k \leq n < 2^{k+1} \Leftrightarrow k = \lceil \log_2 n \rceil$ .

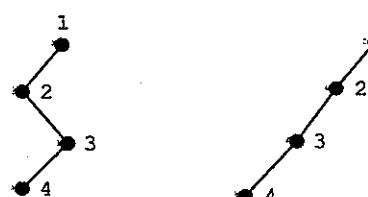
Construim arboarele binar plin cu  $2^{k+1}-1$  noduri și eliminăm de pe ultimul nivel nodurile  $2^{k+1}-1, 2^{k+1}-2, \dots, n+1$ .

**Exemplu**

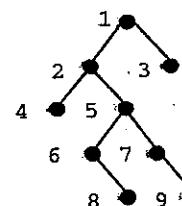
Arboarele binar complet cu 5 vârfuri se obține prin eliminarea vârfurilor 7 și 6 din arboarele binar plin cu înălțimea 2 :

**Arbore binar degenerat**

Un arbore binar se numește degenerat dacă are  $n$  noduri dispuse pe  $n$  niveluri.

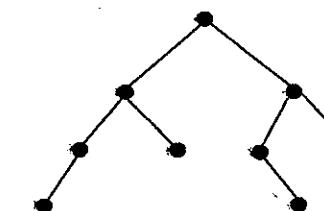
**Exemplu****Exerciții propuse**

1. Se consideră arboarele din figura următoare :

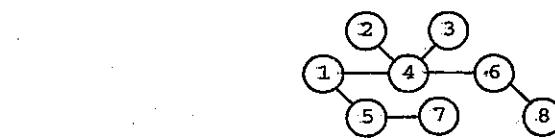


- a. Ce înălțime are acest arbore ?
- b. Care sunt strămoșii nodului 6 ?
- c. Care sunt descendenții nodului 2 ?

- d. Care sunt nodurile terminale în arbore ?
- e. Care sunt descendenții direcți ai nodului 5 ?
- f. Care noduri din arbore nu au fiu stâng ?
- g. Care sunt nodurile situate pe nivelul 2 ?
- h. Dați exemplu de două noduri frați.
  
- 2. Desenați toți arborii binari cu 3 noduri numerotate de la 1 la 3. Câte dintre aceștia sunt stricti ?
- 3. Desenați arboarele binar complet cu 13 vârfuri. Ce înălțime are acest arbore ?
- 4. Desenați toți arborii binari cu 5 noduri dintre care 3 frunze.
- 5. Câte arbori degenerați cu 4 vârfuri există ?
- 6. Demonstrați prin inducție matematică faptul că numărul maxim de noduri de pe nivelul  $i$  al unui arbore binar este  $2^i$ .
- 7. Pe baza rezultatului de la exercițiul 2, demonstrați că numărul maxim de noduri într-un arbore cu înălțimea  $h$  este  $2^{h+1}-1$ .
- 8. Demonstrați că un arbore cu  $n$  vârfuri are înălțimea cel puțin egală cu  $\lceil \log_2 n \rceil$ .
- 9. Se consideră un arbore cu rădăcină cu 17 noduri numerotate de la 1 la 17. Rădăcina este nodul 1. Fiecare nod  $i$  are drept fiu nodurile  $3i-1, 3i$  și  $3i+1$  (dacă acestea există). Ce înălțime are acest arbore ? Care sunt nodurile terminale în arbore ?
- 10. Se consideră un arbore cu înălțimea 3, în care fiecare nod are exact 4 fiu. Câte noduri are în total arborele ?
- 11. Construiți un arbore binar strict cu număr minim de noduri și înălțimea egală cu 4. Câte noduri terminale există în arbore ?
- 12. Ce valoare de adevară are propoziția „orice arbore binar complet este strict” ?
- 13. Se consideră următorul arbore binar. Adăugați un număr minim de noduri astfel încât arboarele să devină complet.



- 14. Se consideră un arbore cu rădăcină în care orice nod care nu este terminal are exact 3 descendenți direcți. Atunci numărul de noduri terminale (frunze) ale arborelui poate fi :
  - a. 14
  - b. 24
  - c. 15
  - d. 2
 (Bacalaureat, iulie 2003)
  
- 15. Se consideră arboarele din figura următoare. Care este înălțimea maximă a arborelui cu rădăcină ce se poate obține alegând ca rădăcina unul dintre noduri ?



- a. 3      b. 6      c. 7      d. 5

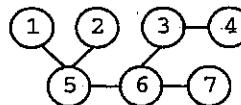
(Bacalaureat, iulie 2003)

16. Un arbore are 14 noduri. Atât rădăcina, cât și fiecare dintre nodurile neterminale au cel puțin 3 descendenți direcți. Înălțimea maximă a arborelui (numărul maxim de muchii ce leagă rădăcina de o frunză) este:

- a. 4      b. 5      c. 3      d. 13

(Bacalaureat special, 2003)

17. Se consideră arborele din figura următoare. Care dintre noduri trebuie alese ca rădăcină astfel încât înălțimea arborelui să fie minimă?



- a. 6      b. 5      c. 3      d. 7

(Simulare Bacalaureat, 2003)

### 3.3. Reprezentarea arborilor cu rădăcină

#### *Reprezentarea cu referințe descendente*

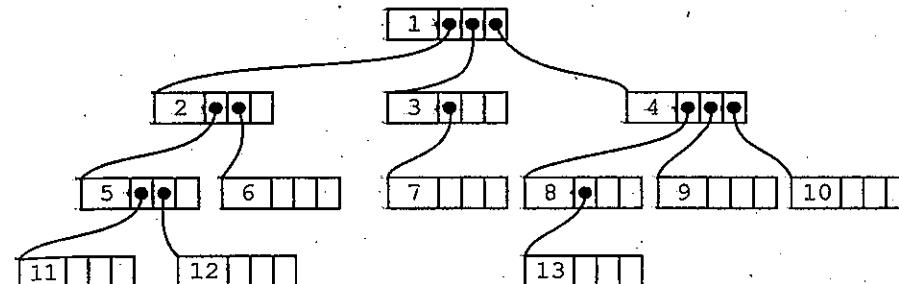
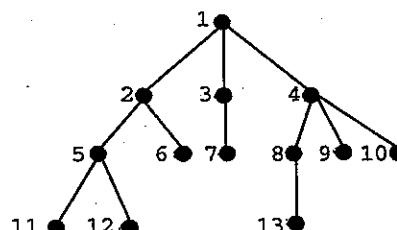
Pentru fiecare nod din arbore vom reține informația asociată nodului, precum și referințe către fiili săi. Dacă presupunem că gradul oricărui nod este cel mult egal cu NrMaxFii, putem reprezenta referințele spre fiil printr-un vector cu NrMaxFii componente. Declararea structurii ce reprezintă un nod din arbore va fi:

```

struct Nod
{
    Tip_inf inf;
    struct Nod * Fiu[NrMaxFii];
};
  
```

#### *Exemplu*

Reprezentarea cu referințe descendente a arborelui din figura următoare, în care fiecare nod are maxim 3 fiil este:



Când arborele este reprezentat prin referințe descendente este suficient să reținem rădăcina arborelui pentru a avea acces la toate nodurile acestuia.

În declarația precedentă, vectorul de fiil este alocat static. Pentru a evita o risipă excesivă de memorie este recomandabil ca vectorul de fiil să fie alocat dinamic și să adăugăm un câmp suplimentar în care să reținem numărul de fiil ai nodului :

```

struct Nod
{
    Tip_inf inf;
    int nr_fii;
    struct Nod * * Fiu; };
  
```

în cazul în care arborele cu rădăcină este binar, pentru fiecare nod vom reține adresa fiului stâng și adresa fiului drept :

```

struct Nod
{
    Tip_inf inf;
    struct Nod * St, * Dr;
};
  
```

#### *Reprezentarea cu referințe ascenđante*

În acest mod de reprezentare, pentru fiecare nod din arbore reținem, pe lângă informația aferentă nodului, o referință către nodul său părinte. Cea mai simplă modalitate de a implementa reprezentarea cu referințe ascenđante este de a considera doi vectori (intr-un vector memorăm informațiiile asociate nodurilor, iar în celălalt vector memorăm pentru fiecare nod din arbore indicele nodului său părinte).

#### *Exemplu*

Considerăm arborele din exemplul precedent. Informațiiile asociate nodurilor sunt numerele de la 1 la 13 și coincid cu indicii nodurilor din vector. Vectorul tata, în care reținem indicele nodului părinte al fiecarui nod, va fi :

tata =	0	1	1	1	2	2	3	4	4	4	5	5	8
	1	2	3	4	5	6	7	8	9	10	11	12	13

#### *Reprezentarea Fiul-Frate*

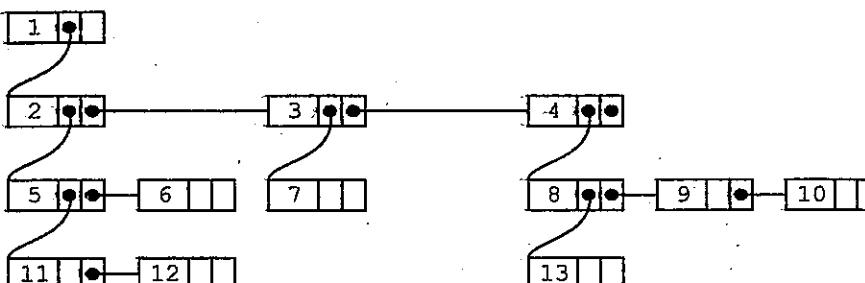
Pentru fiecare nod din arbore reținem fiul cel mai din stânga și fratele lui din dreapta cel mai apropiat.

```

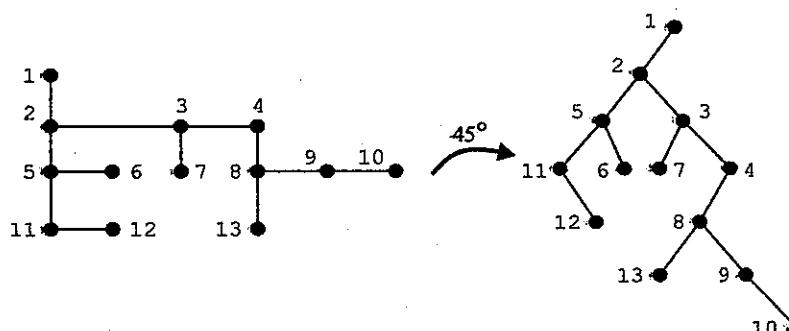
struct Nod
{
    Tip_inf inf;
    struct Nod * Fiu, * Frate;
};


```

Pentru arborele din exemplul precedent obținem următoarea reprezentare fiu-frate:



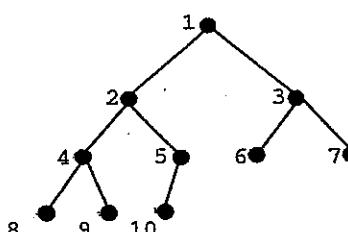
Rotind această reprezentare cu  $45^\circ$  în sensul acelor de ceasornic, obținem un arbore binar în care pentru fiecare nod, fiul drept este fratele lui din dreapta cel mai apropiat. Figura de mai jos ilustrează această transformare.



### Reprezentarea arborilor binari compleți

Să considerăm un arbore binar complet în care rădăcina este numerotată cu 1, fiul stâng al rădăcinii cu 2, fiul drept al rădăcinii cu 3 și aşa mai departe. Cele  $2^i$  noduri de pe nivelul  $i$  sunt numerotate  $2^i, 2^i+1, \dots, 2^{i+1}-1$  de la stânga la dreapta. Pe ultimul nivel, nodurile sunt numerotate până la  $n$ , numărul de noduri din arbore.

#### Exemplu



Considerând această numerotare a nodurilor, relațiile dintre noduri sunt implicate. Astfel, pentru orice nod  $x$  din arbore:

1. fiul stâng al lui  $x$  este  $2 \cdot x$  (dacă  $2 \cdot x \leq n$ );
2. fiul drept al lui  $x$  este  $2 \cdot x + 1$  (dacă  $2 \cdot x + 1 \leq n$ );
3. nodul părinte al lui  $x$  este  $x/2$  (dacă  $x > 1$ ).

Prin urmare este suficient să reținem într-un vector doar informațiile asociate nodurilor arborelui. O astfel de reprezentare este denumită *reprezentare secvențială*.

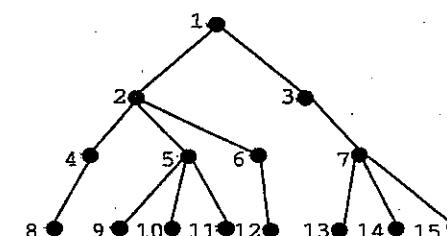
### Exerciții propuse

1. Un arbore are nodurile numerotate de la 1 la 5. Care poate fi vectorul de tați corespunzător?

- a. 5 4 2 1 3      b. 5 2 4 5 0      c. 2 4 0 3 4      d. 2 1 0 3 4

(Simulare Bacalaureat, 2003)

2. Se consideră arboarele cu rădăcină din figura următoare. Ilustrați reprezentarea cu referințe descendente, reprezentarea cu referințe ascendente și reprezentarea fiu-frate corespunzătoare acestui arbore.



3. Se consideră un arbore cu următoarele proprietăți: rădăcina este nodul 1 și fiecare nod  $i$  ( $1 \leq i \leq 3$ ) aflat pe nivelul  $j$  are ca descendenți nodurile  $i+j+1$  și  $i+2 \cdot (j+1)$ . Nodurile  $i$  ( $i > 3$ ) sunt noduri terminale. Stabilii care dintre vectorii următori este vectorul de tați corespunzător arborelui:

- a. 0 1 1 2 3 2 3      b. 0 1 2 2 1 3 3  
c. 0 1 1 2 2 3 3      d. 0 -1 1 -1 -1 1 1

(Simulare Bacalaureat, 2003)

### 3.4. Crearea unui arbore binar

Mecanismul de creare a unui arbore binar depinde în mod esențial de tipul arborelui. Vom prezenta pentru început două proceduri de creare, urmând ca la fiecare tip de arbore studiat să specificăm modul de creare.

Vom reprezenta arboarele binar prin referințe descendente.

```

struct Nod
{
    int inf;
    struct Nod * st, * dr; };
typedef struct Nod * Arbore;
Arbore x;

```

### *Crearea interactivă a unui arbore binar*

Vom crea un arbore binar în mod interactiv, structura arborelui fiind stabilită pe baza răspunsurilor obținute de la utilizator.

Mai exact, pentru fiecare nod din arbore, programul va solicita informația asociată nodului (pe care noi o vom considera pentru simplitate un număr întreg), va întreba dacă nodul are fiu stâng (în caz afirmativ se creează în același mod subarborele stâng), apoi va întreba dacă nodul are fiu drept (în caz afirmativ se va crea în același mod subarborele drept).

Procedeul de creare este descris într-o manieră recursivă, prin urmare vom implementa o funcție recursivă de creare care va returna ca parametru adresa rădăcinii arborelui creat.

```

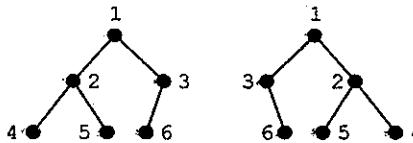
Arbore Creare()
{
    char c;
    Arbore p=new Nod;
    cout<<"Informatia nodului: "; cin>>p->inf;
    cout<<"Exista fiu stang? (d/n) "; cin>>c;
    if (c=='d') p->st=Creare();
    else p->st=NULL;
    cout<<"Exista fiu drept? (d/n) "; cin>>c;
    if (c=='d') p->dr=Creare();
    else p->dr=NULL;
    return p;
}

```

### *Crearea unui arbore binar echilibrat după numărul de noduri*

Un arbore binar se numește echilibrat după numărul de noduri dacă pentru orice nod din arbore, diferența dintre numărul nodurilor din subarborele stâng și numărul nodurilor din subarborele drept este cel mult o unitate.

#### *Exemple*



Pentru a crea un arbore binar echilibrat cu  $n$  vârfuri se stabilește un vârf rădăcină. Se construiește un arbore binar echilibrat cu  $n/2$  vârfuri ca subarbore stâng și un

arbore binar echilibrat cu  $n-1-n/2$  vârfuri ca subarbore drept. Dacă  $n$  este par numărul nodurilor din subarborele stâng va fi cu o unitate mai mare decât numărul nodurilor din subarborele drept, altfel cei doi subarbore au un număr egal de noduri.

```

Arbore Creare(int n)
{
    /* returneaza adresa rădăcinii unui arbore binar echilibrat
       cu n noduri */
    if (n)
    {
        Arbore p=new Nod;
        cout<<"Informatia nodului: "; cin>>p->inf;
        p->st=Creare(n/2);
        p->dr=Creare(n-1-n/2);
        return p;
    }
    return NULL;
}

```

### **3.5. Parcugerea arborilor binari**

Parcugerile sunt cel mai frecvent utilizate operații pe arbori. A parcurge un arbore înseamnă a vizita fiecare nod al arborelui o singură dată, în scopul prelucrării informației asociate nodului.

Există mai multe posibilități de parcugere a arborilor – în adâncime (preordine, inordine, postordine) sau pe niveluri.

#### *Parcugerile în adâncime*

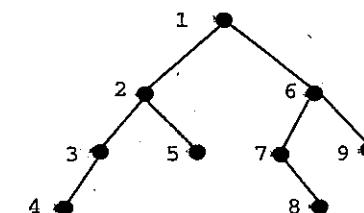
În toate cele trei tipuri de parcugere în adâncime se vizitează mai întâi subarborele stâng, apoi subarborele drept. Diferența constă în poziția rădăcinii față de cei doi subarbore.

##### *Parcugerea preordine*

Pentru a parcurge un arbore binar în preordine se vizitează mai întâi rădăcina, se parcurge în preordine subarborele stâng, apoi se parcurge în preordine subarborele drept.

##### *Exemplu*

Parcugerea preordine a arborelui următor este : 1, 2, 3, 4, 5, 6, 7, 8, 9.



Funcția preordine are ca parametru adresa rădăcinii arborelui ce urmează să fie parcurs și afișează pe ecran informațiile nodurilor arborelui.

```
void preordine(Arbore r)
{
    if (r)
        {cout<<r->inf<<' ';
        preordine(r->st);
        preordine(r->dr);
    }
}
```

#### Parcursarea inordine

Pentru a parcurge în inordine un arbore binar se parcurge în inordine subarborele stâng, se vizitează rădăcina, apoi se parcurge în inordine subarborele drept.

#### Exemplu

Parcursarea inordine a arborelui din exemplul precedent este : 4, 3, 2, 5, 1, 7, 8, 6, 9.

```
void inordine(Arbore r)
{
    if (r)
        {inordine(r->st);
        cout<<r->inf<<' ';
        inordine(r->dr);
    }
}
```

#### Parcursarea postordine

Pentru a parcurge în postordine un arbore binar se parcurge în postordine subarborele stâng, apoi cel drept, apoi se vizitează rădăcina.

#### Exemplu

Parcursarea postordine a arborelui din exemplul precedent este : 4, 3, 5, 2, 8, 7, 9, 6, 1.

```
void postordine(Arbore r)
{
    if (r)
        {postordine(r->st);
        postordine(r->dr);
        cout<<r->inf<<' ';
    }
}
```

#### Parcursarea pe niveluri

Pentru a parcurge pe niveluri un arbore binar se vizitează întâi rădăcina, apoi fiul stâng al rădăcinii, apoi cel drept și se continuă în acest mod, vizitând nodurile de pe fiecare nivel de la stânga la dreapta. Acest mod de parcursare corespunde unei parcursări în lățime a grafului orientat corespunzător, începând din rădăcină.

#### Exemplu

Parcursarea pe niveluri a arborelui din exemplul precedent este : 1, 2, 6, 3, 5, 7, 9, 4, 8.

### 3.6. Determinarea înălțimii unui arbore

Dacă arborele este vid, prin convenție, se consideră că înălțimea sa este -1. Astfel, putem calcula înălțimea arborelui ca fiind maximul dintre înălțimile subarborelor rădăcinii la care se adaugă nivelul pe care se află rădăcina.

```
int inaltime(Arbore r)
//întoarce înălțimea arborelui binar cu radacina r
{
    if (!r) return -1;
    int hs=inaltime(r->st);
    int hd=inaltime(r->dr);
    return 1+(hs>hd?hs:hd);
}
```

### 3.7. Crearea unui arbore binar pe baza parcurgerilor în preordine și inordine

Se dă secvențele obținute prin parcurgerile în preordine și în inordine ale unui arbore binar. Să se reconstituie arborele binar corespunzător.

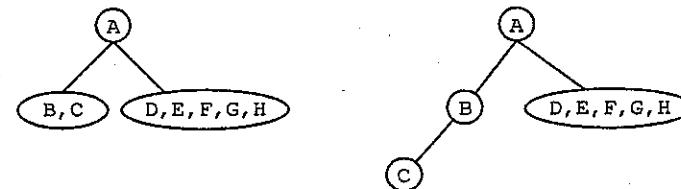
#### Exemplu

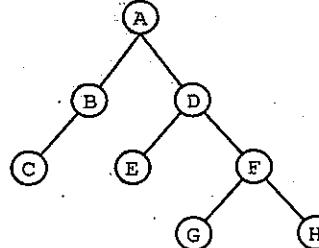
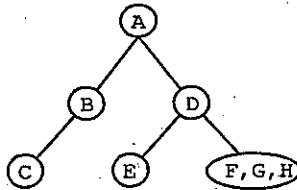
Fie A, B, C, D; E, F, G, H secvența informațiilor nodurilor obținută prin parcursarea în preordine și C, B, A, E, D, G, F, H secvența informațiilor nodurilor obținută prin parcursarea în inordine.

Analizând parcursarea în preordine, deducem că nodul A este rădăcina. De asemenea, analizând parcursarea în inordine, deducem că nodurile C, B constituie arborele stâng, iar D, E, G, F, H subarborele drept.

Pentru a determina structura subarborelui stâng, respectiv a subarborelui drept, procedăm în același mod : din parcursarea în preordine deducem că B este rădăcina subarborelui stâng, iar din parcursarea în inordine deducem că C este fiul stâng al lui B. În mod similar, pentru subarborele drept deducem din parcursarea în preordine că D este rădăcină, iar din parcursarea în inordine că subarborele stâng al lui D este format numai din nodul E, iar subarborele drept al lui D din nodurile F, G, H. Procedeul se repetă până când obținem întreg arborele.

Succesiunea operațiilor este ilustrată în figura următoare :





### Propoziție

Succesiunile de noduri obținute prin parcurgerile în inordine și în preordine ale unui arbore binar definesc în mod unic structura arborelui.

### Demonstratie

Vom proceda prin inducție completă după numărul de noduri. Dacă arborele are un singur nod, rădăcina, afirmația este evidentă. Presupunem că pentru  $\forall k \in \{1, 2, \dots, n\}$  afirmația este adeverată, adică pentru orice pereche de secvențe inordine-preordine de lungime  $k$ , arborele binar corespunzător este unic. Să demonstrăm că orice pereche de secvențe preordine-inordine de lungime  $n+1$  determină în mod unic un arbore binar.

Să considerăm o pereche de secvențe preordine-inordine de lungime  $n+1$ . Primul nod din parcurgerea în preordine este în mod necesar rădăcina arborelui, celelalte  $n$  noduri fiind distribuite în subarbore: toate nodurile situate în stânga rădăcinii în parcurgerea în inordine vor constitui subarborele stâng, nodurile situate în dreapta rădăcinii în parcurgerea inordine vor constitui subarborele drept.

Obținem două perechi de secvențe preordine-inordine de lungime cel mult  $n$ , care, din ipoteza inductivă, determină în mod unic subarborele stâng, respectiv pe cel drept și, în consecință, cum rădăcina este în mod unic determinată, deducem că perechea de secvențe preordine-inordine de lungime  $n+1$  determină în mod unic un arbore binar cu  $n$  noduri.

Vom descrie o funcție recursivă `Creare()` care determină arborele binar corespunzător unei perechi de secvențe preordine-inordine date. Pentru simplificare vom considera că nodurile arborelui sunt numerotate în preordine de la 1 la  $n$ . Astfel, este suficient să reținem într-un vector global în indicii vîrfurilor în ordinea în care au fost atinse în inordine. Inițial, apelăm `Creare(1, 1, n)`.

```

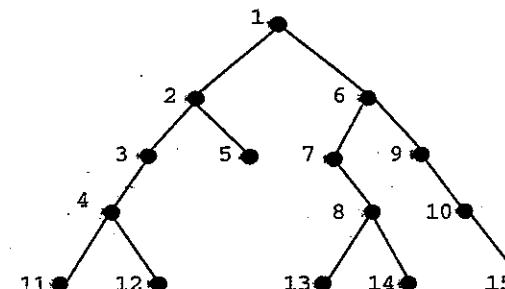
Arbore Creare (int rad, int st, int dr)
/* functia returneaza radacina arborelui unic determinat de
parcurgerile, inordine-preordine; rad este indicele radacinii
arborelui, st si dr sunt limitele intre care se gaseste
parcurgerea inordine a arborelui in vectorul in */
{int IPozRad;
Arbore r=new Nod; //alloc memorie pentru radacina
r->inf=rad; //retinem informatia
// determin pozitia radacinii in parcurgerea inordine
for (IPozRad=st; in[IPozRad]!= rad; IPozRad++);
}
  
```

```

if (IPozRad==st) //subarborele stang este vid
  r->st=NULL;
else
  // in[st.. IPozRad-1] contine subarborele stang
  r->st=Creare(rad+1, st, IPozRad-1);
if (IPozRad==dr) //subarborele drept este vid
  r->dr=NULL;
else
  // in[IPozRad+1.. dr] contine subarborele drept
  r->dr=Creare(rad+IPozRad-st+1, IPozRad+1, dr);
  // in subarborele stang au fost IPozRad-st+1 varfuri
return r;
}
  
```

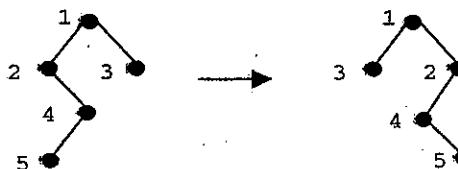
### Exerciții propuse

- Considerăm arborele din figura următoare. Parcurgeți acest arbore în inordine, preordine, postordine și pe niveluri.



- Să considerăm un arbore cu 15 vîrfuri numerotate în preordine de la 1 la 15. Parcurgerea inordine a acestui arbore este:  
4 3 5 2 7 6 8 1 9 11 12 13 15 14 10  
Reconstituji arborele.
- Ilustrați grafic un arbore binar echilibrat după numărul de noduri, având 8 noduri, obținut prin apelarea funcției de creare prezentată.
- Scrieți o funcție care să creeze un arbore binar complet în reprezentarea cu referințe descendente. Nodurile arborelui vor fi numerotate în ordinea nivelurilor, de la stânga la dreapta, începând cu 1 pentru rădăcină.
- Scrieți o funcție care să testeze dacă doi arbori binari primiți ca parametru sunt egali. Spunem că doi arbori binari sunt egali dacă au aceeași structură și conțin același informații în nodurile corespondente.
- Scrieți o funcție care să creeze o copie a unui arbore binar.
- Scrieți o funcție care să parcurgă pe niveluri un arbore binar specificat ca parametru.
- Scrieți o funcție care să numere câte noduri terminale are un arbore binar primit ca parametru.

9. Scrieți o funcție de stergere a unui arbore binar (funcția trebuie să elibereze zonele de memorie alocate nodurilor arborelui).
10. Scrieți o funcție care să caute într-un arbore binar un nod ce conține ca informație o valoare dată  $x$ .
11. Scrieți o funcție care să verifice dacă un arbore binar primit ca parametru este strict.
12. Scrieți o funcție care să verifice dacă un arbore binar primit ca parametru este echilibrat după numărul de noduri.
13. Scrieți o funcție care pentru un arbore binar primit ca parametru schimbă fiul stâng cu fiul drept pentru orice nod din arbore. De exemplu :



14. Să se determine numărul de arbori binari distincți cu  $n$  noduri, făcând abstracție de numerotarea nodurilor. De exemplu, pentru  $n=3$  există 5 arbori binari distincți cu 3 noduri.
15. Demonstrați că secvența obținută prin parcurgerea postordine a arborelui și secvența obținută prin parcurgerea inordine a arborelui determină în mod unic arborele. Scrieți o funcție care să reconstituie arborele pe baza parcurgerilor postordine-inordine.
16. Scrieți o funcție care să parcurgă postordine un arbore cu rădăcină în reprezentarea fiu-frate.
17. Scrieți o funcție care să determine înălțimea unui arbore cu rădăcină în reprezentarea fiu-frate.
18. Scrieți o funcție care să determine înălțimea unui arbore cu rădăcină în reprezentarea cu referințe descendente.

### 3.8. Heap-uri

Situații practice impun în mod frecvent alegerea dintr-o mulțime dinamică de valori a uneia sau a uneia care îndeplinește anumite condiții. Spre exemplu, o firmă are în vedere spre onorare, în primul rând, comenziile cele mai rentabile. Este deci necesar ca o structură de date dinamică adecvată să ofere cu „efort” minim de prelucrare, informația cerută de un anumit criteriu de optim. Constatăm că este vorba despre *selectarea* unor informații dintr-un volum de date, organizate după un criteriu stabilit. O astfel de facilitate de prelucrare este oferită de o categorie de *arbori binari*, cunoscuți în literatura de specialitate sub numele de *heap-uri*.

Un *max-heap* este un arbore binar complet în care valoarea memorată în orice nod al său este mai mare sau egală cu valorile memorate în fiili săi.

În mod analog, se poate defini *min-heap*-ul ca un arbore binar complet în care valoarea memorată în orice nod este mai mică sau egală cu valorile memorate în nodurile fiilor acestuia.

În cele ce urmează, un *max-heap* va fi numit *heap*, urmând ca atunci când este vorba despre un *min-heap*, aceasta să se precizeze în mod explicit.

Fiind un arbore binar complet, reprezentarea cea mai adecvată a unui *heap* este reprezentarea secvențială. Dacă *heap*-ul are  $n$  noduri numerotate de la 1 la  $n$ , va fi suficient să reținem informațiile asociate nodurilor într-un vector cu  $n$  elemente, relațiile dintre noduri fiind implicate.

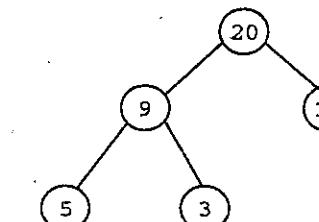
Utilizând reprezentarea secvențială, putem reformula definiția unui *heap* astfel :

Tabloul  $H_n$  formează un *max-heap* dacă  $H[i] \leq H[i/2]$ ,  $\forall i \in \{2, 3, \dots, n\}$ .

Tabloul  $H_n$  formează un *min-heap* dacă  $H[i] \geq H[i/2]$ ,  $\forall i \in \{2, 3, \dots, n\}$ .

#### Exemplu

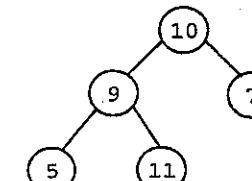
Arborele binar complet din figură este un *max-heap*. Reprezentarea sa secvențială este : 20 9 17 5 3.



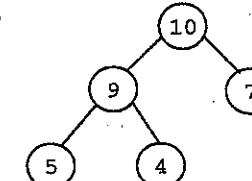
#### Exerciții propuse

1. Care dintre următorii arbori reprezintă un *max-heap* ?

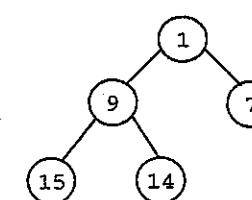
a.



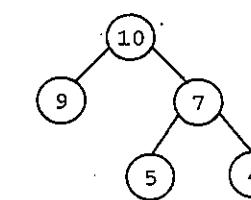
b.



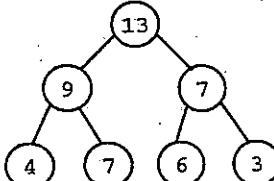
c.



d.



2. Să considerăm următorul vector:  $H=(2, 5, 3, 9, 8, 4, 10, 20, 10, 9)$ . Poate fi considerat reprezentarea secvențială a unui *min-heap*?  
 3. Un vector ordonat crescător poate constitui reprezentarea implicită a unui *heap*?  
 4. Se consideră *max-heap*-ul următor:



Care dintre următorii vectori este reprezentarea secvențială a acestui *max-heap*?

- a.  $H=(13, 9, 4, 7, 7, 6, 3)$ ;      b.  $H=(13, 9, 7, 4, 7, 6, 3)$ ;  
 c.  $H=(13, 7, 9, 3, 6, 7, 4)$ ;      d.  $H=(9, 4, 7, 7, 6, 3, 13)$ .

5. Scrieți o funcție care să verifice dacă un vector specificat ca parametru este *max-heap*.

### Crearea unui heap prin inserări successive

O primă soluție este de a insera succesiv elementele în *heap*, plecând inițial de la *heap*-ul format dintr-un singur element. Funcția *CreareHeap()* organizează ca *heap* vectorul global *H*, inserând succesiv cele *n* elemente.

```
void CreareHeap()
{
    int i;
    for (i=2; i<=n; i++)
        InsertHeap(i-1, H[i]);
}
```

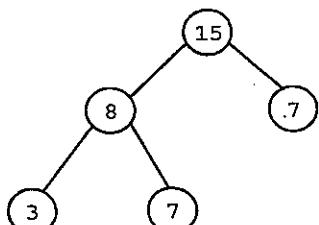
### Inserarea unui nod într-un heap

Fie  $H_n$  un *heap* cu *n* elemente și *x* valoarea ce trebuie inserată. Valoarea *x* va fi memorată în *heap* pe poziția *n+1*, apoi se va restaura *heap*-ul, „promovând” valoarea *x* spre rădăcină, până când proprietatea de *heap* este restabilită.

#### Exemplu

Să inserăm valoarea 20 în *heap*-ul din figură:

Reprezentare arborescentă:

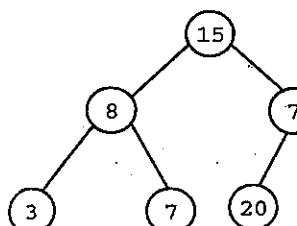


Reprezentare secvențială:

$H=$	15	8	7	3	7	
	1	2	3	4	5	

Valoarea inserată va ocupa poziția 6.

Reprezentare arborescentă:

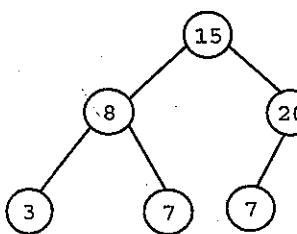


Reprezentare secvențială:

$H=$	15	8	7	3	7	20
	1	2	3	4	5	6

Acest arbore binar complet nu este un *heap*. Vom compara valoarea nodului 6 cu valoarea nodului său părinte (nodul 3). Pentru a conserva proprietatea de *heap* trebuie să interschimbăm valorile celor două noduri.

Reprezentare arborescentă:

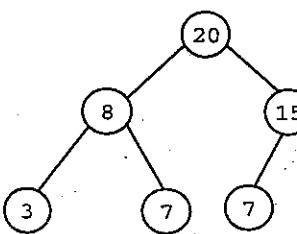


Reprezentare secvențială:

$H=$	15	8	20	3	7	7
	1	2	3	4	5	6

Din nou, valoarea 20 va fi comparată cu valoarea din nodul său părinte (nodul 1) și, deoarece nu respectă condiția de *max-heap*, va fi interschimbată.

Reprezentare arborescentă:



Reprezentare secvențială:

$H=$	20	8	15	3	7	7
	1	2	3	4	5	6

Valoarea 20 a fost promovată până în rădăcină, valorile de pe drumul parcurs spre rădăcină fiind retrogradeate succesiv, cu un nivel. Proprietatea de *heap* a fost restabilită.

```
void InsertHeap (int n, int x)
//inserăza valoarea x în heap-ul H de dimensiune n
{
    int fiu=++n;
    //fiu indică poziția pe care trebuie plasata valoarea x
    int tata=n/2;
```

```

while (tata && H[tata]<x)
    // valoarea x trebuie "promovată"
    {H[fiu]=H[tata];
     fiu=tata;
     tata=fiu/2;
    }
H[fiu]=x;
}

```

#### Observație

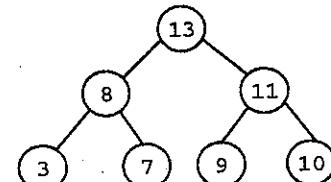
Analizând complexitatea procedurii de inserare deducem că, în cazul cel mai favorabil, valoarea nou inserată urcă până în rădăcina arborelui, deci necesită  $O(n)$  operații elementare, unde  $n$  este înălțimea *heap*-ului. Fiind un arbore binar complet, înălțimea *heap*-ului  $h$  va fi  $\lceil \log_2 n \rceil$ , deci inserarea unui nod într-un *heap* cu  $n$  elemente are complexitatea timp  $O(n \log n)$ .

Să estimăm complexitatea, în cazul cel mai favorabil, a procedurii de construcție a *heap*-ului prin inserări succesive. Dacă toate cele  $2^i$  valori de pe nivelul  $i$  urcă până în rădăcină, obținem :

$$\sum_{i=1}^{\lceil \log n \rceil} i \cdot 2^i < \log n \sum_{i=1}^{\lceil \log n \rceil} 2^i = O(n \log n)$$

#### Exerciții propuse

1. Să se insereze valoarea 15 în *max-heap*-ul următor :



2. Vectorul  $H=(3, 5, 7, 5, 8, 10, 13, 20, 6, 40, 10, 11, 12, 15, 19)$  este reprezentarea secvențială a unui *min-heap*. Inserați în *min-heap* valoarea 4.
3. Creați un *max-heap* inserând succesiv valorile: 7, 20, 5, 10, 13, 8, 11, 9, 15, 17, 13.
4. Implementați funcția de inserare a unei valori într-un *min-heap*.

#### Crearea unui *heap* prin combinări succesive

O strategie mai eficientă de construcție a unui *heap* se bazează pe ideea de echilibrare. Apelul funcției *InsertHeap*(n, x) poate fi interpretat ca o combinare de două *heap*-uri: un *heap* cu  $n$  elemente și un *heap* format numai din elementul  $x$ .

Putem construi *heap*-ul cu rădăcina  $H[i]$  combinând la fiecare pas î două *heap*-uri de dimensiuni apropiate: *heap*-ul cu rădăcina  $2i$  cu *heap*-ul cu rădăcina  $2i+1$  și cu elementul  $H[i]$ .

```

void CreareHeap()
{
    int i;
    for (i=n/2; i; i--)
        CombHeap(i, n);
}

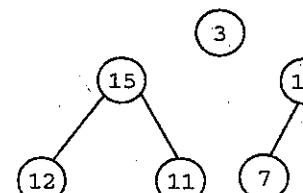
```

Funcția *CombHeap*(i, n) combină elementul  $H[i]$  cu *heap*-ul cu rădăcina  $2i$  și cu *heap*-ul cu rădăcina  $2i+1$ .

#### Exemplu

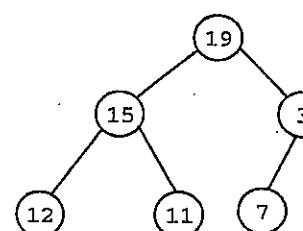
Să analizăm combinarea *heap*-urilor cu rădăcina în nodul 2, respectiv 3, cu nodul de pe poziția 1 (având valoarea 3) :

Reprezentare arborescentă :



Valoarea din nodul 1 (3) va fi comparată cu cel mai mare dintre fii (nodul 3, având valoarea 19). Deoarece  $3 < 19$ , pentru a îndeplini condiția de *heap*, valorile nodurilor 1 și 3 vor fi interschimbate :

Reprezentare arborescentă :



Valoarea 3 a fost „retrogradată” în arbore, este acum plasată în nodul 3 și urmează să fie comparată cu cel mai mare dintre fiii nodului 3 (nodul 6, având valoarea 7). Condiția de *max-heap* nu este îndeplinită, prin urmare, valoare nodului 3 va fi interschimbată cu valoarea nodului 6 :

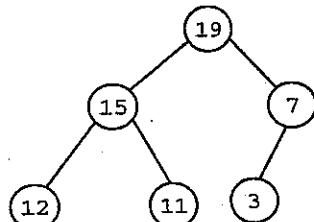
Reprezentare secvențială :

H=	3	15	19	12	11	7
	1	2	3	4	5	6

Reprezentare secvențială :

H=	19	15	3	12	11	7
	1	2	3	4	5	6

Reprezentare arborescentă:



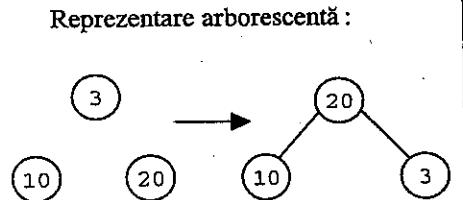
Am obținut astfel un *heap*, prin combinarea a două *heap*-uri.

```
void CombHeap (int i, int n)
{
    int v=H[i], tata=i, fiu=2*i;
    //tata indică pozitia pe care va fi plasată valoarea v
    while (fiu<=n)
        if (fiu<n) //există fiu drept
            if (H[fiu]<H[fiu+1]) fiu++;
        //fiu indică fiul cu valoarea cea mai mare
        if (v<H[fiu])
            H[tata]=H[fiu];
            tata=fiu; //v coboară în arbore
            fiu=fiu*2;
        else
            fiu=n+1; //am gasit pozitia corectă pentru v
    }
    H[tata]=v;
}
```

#### Exemplu

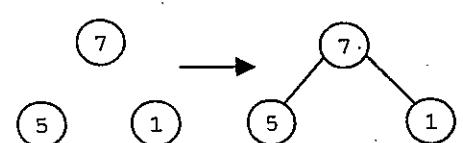
Pentru organizarea ca *heap* a vectorului  $H = \{0, 7, 3, 5, 1, 10, 20\}$  se vor executa 3 pași. La pasul  $i=3$ , se apelează *CombHeap*(3, 7):

Reprezentare arborescentă:



La pasul  $i=2$ , se apelează *CombHeap*(2, 7):

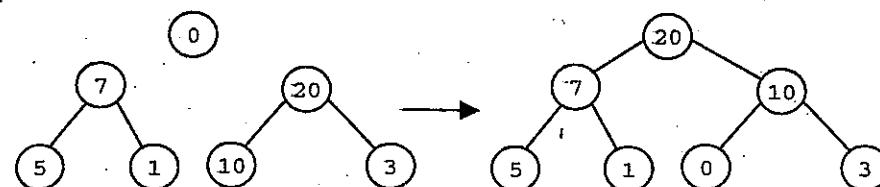
Reprezentare arborescentă



Reprezentare secvențială:

$H =$	19	15	7	12	11	3
	1	2	3	4	5	6

Observați că nu s-a efectuat nici o interschimbare la pasul  $i=2$ . La pasul  $i=1$  se apelează *CombHeap*(1, 7):



Reprezentarea secvențială a *heap*-ului este:

$H =$	20	7	10	5	1	0	3
	1	2	3	4	5	6	7

#### Complexitatea algoritmului

Notăm cu  $h = \lceil \log_2 n \rceil$  înălțimea *heap*-ului. Pentru fiecare nod  $x$  de pe nivelul  $i$ ,  $i \in \{0, 1, \dots, h-1\}$ , pentru a construi *heap*-ul cu rădăcina  $x$  se fac cel mult  $h-i$  coborări ale valorii  $x$  în arbore.

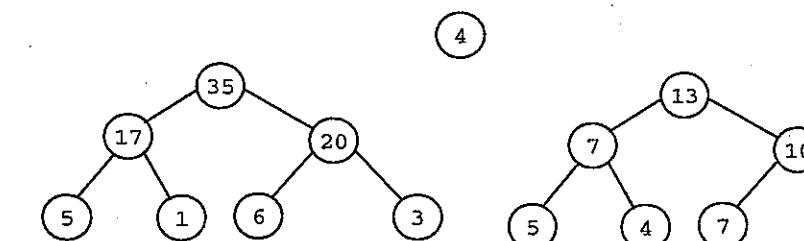
Prin urmare, în cazul cel mai defavorabil, numărul de operații elementare efectuate de algoritm pentru a construi un *heap* cu  $n$  elemente este:

$$T_n \leq \sum_{i=0}^{h-1} 2^i (h-i) = \sum_{j=1}^h j \cdot 2^{h-j} = \sum_{j=1}^h 2^h \cdot \frac{j}{2^j} \leq n \sum_{j=1}^h \frac{j}{2^j} < 2n = O(n)$$

Algoritmul de construcție a unui *heap* a devenit astfel liniar.

#### Exerciții propuse

- Să se combine *heap*-urile din figura următoare cu nodul având valoarea 4:



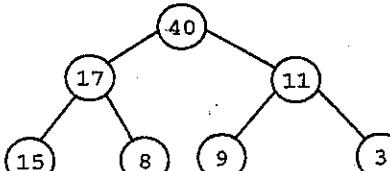
- Să transforme într-un *min-heap* vectorul  $H = \{14, 5, 20, 9, 22, 1, 0, 7, 23, 11, 8, 2, 7, 4\}$ , prin combinări succesive de *min-heap*-uri.
- Implementați funcția de combinare a două *min-heap*-uri.

### Extragerea valorii maxime dintr-un max-heap

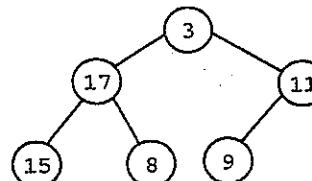
Într-un *max-heap* valoarea maximă este plasată în rădăcina arborelui. Pentru a extrage din *heap* valoarea maximă vom înlocui valoarea rădăcinii cu valoarea ultimului element din *heap*, eliminăm ultimul element din *heap*, apoi restaurăm *heap*-ul combinând *heap*-ul având rădăcina în nodul 2, cu *heap*-ul având rădăcina în nodul 3 și cu elementul plasat în rădăcină.

#### Exemplu

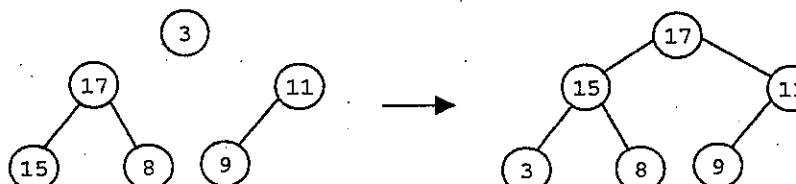
Să considerăm *heap*-ul următor :



Pentru a extrage maximumul plasăm valoarea ultimului nod (3) în rădăcină și eliminăm ultimul nod :



Apoi restaurăm *heap*-ul prin combinarea *heap*-urilor cu rădăcinile în nodurile 2, respectiv 3 cu nodul rădăcină :



```

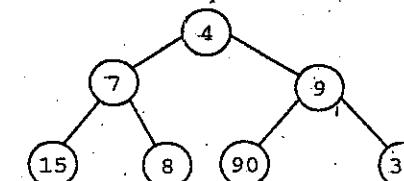
int ExtractMax()
{
    int v=H[1]; //retin valoarea din radacina
    H[1]=H[n--]; //elimin ultimul nod din heap
    CombHeap(1, n); //restaurez heap-ul
    return v;
}
  
```

#### Complexitatea algoritmului

Complexitatea operației de extragere a elementului de cheie maximă se reduce la complexitatea algoritmului de combinare a două *heap*-uri –  $O(\log n)$ .

### Exerciții propuse

1. Să se extragă minimumul din *min-heap*-ul următor :



2. Vectorul  $H=(20, 13, 9, 8, 5, 4, 3, 7, 5, 2)$  este reprezentarea secvențială a unui *max-heap*. Să se extragă maximumul din acest *max-heap*.

### Heapsort

O primă aplicație este sortarea unui vector cu ajutorul *heap*-urilor, metodă cunoscută sub denumirea de *heapsort*.

Primul pas este de a organiza vectorul ca un *heap*. Deoarece, conform proprietății de *heap*, elementul maxim din vector este plasat în rădăcina *heap*-ului, deci pe poziția 1, el poate fi plasat pe poziția sa corectă, interschimbându-l cu elementul de pe poziția  $n$ . Noul element din rădăcina *heap*-ului nu respectă proprietatea de *heap*, dar subarborei rădăcinii rămân *heap*-uri. Prin urmare, trebuie restaurat *heap*-ul apelând funcția *CombHeap(1, n-1)*. Elementul de pe poziția  $n$  – fiind deja pe poziția sa corectă – nu mai este inclus în *heap*.

Procedeul se repetă până când toate elementele vectorului sunt plasate pe pozițiile lor corecte.

Funcția *HeapSort()* ordonează crescător vectorul global  $H$  cu  $n$  elemente :

```

void HeapSort()
{
    int aux, i;
    CreareHeap();
    for (i=n; i>1; i--)
        {aux=H[1]; //interschimba elementul H[i] cu H[1]
        H[1]=H[i];
        H[i]=aux;
        CombHeap(1, i-1); //restaureaza heap-ul
    }
}
  
```

#### Complexitatea algoritmului

Complexitatea algoritmului *heapsort* este  $O(n \log n)$ , crearea *heap*-ului fiind liniară, iar fiecare dintre cele  $n-1$  apeluri ale funcției *CombHeap()* având complexitatea  $O(\log n)$ . Deci algoritmul de sortare cu ajutorul *heap*-urilor este optimă în cazul cel mai defavorabil.

## Cozi cu prioritate

Cel mai frecvent utilizată aplicație a unui *heap* este implementarea cozilor cu prioritate.

O *coadă cu prioritate* este o structură de date abstractă formată din elemente care au asociată o valoare numită cheie sau prioritate și care suportă două operații:

- *Insert(Q, x)*: inserează elementul  $x$  în coadă cu prioritate  $Q$ ;
- *ExtractMax(Q)*: extrage din coadă cu prioritate  $Q$  elementul de valoare maximă și returnează valoarea acestui element.

### Observație

În mod analog, se poate defini o coadă cu min-prioritate pentru care interesează operația de extragere din coadă a elementului de prioritate minimă.

Există multe aplicații ale cozilor cu prioritate. De exemplu, planificarea execuției programelor pe un calculator: coada cu prioritate reține programele ce trebuie executate în funcție de prioritățile lor relative. Când se încheie sau se întrerupe execuția unui program, programul cu cea mai mare prioritate din coadă este selectat și lansat în execuție (operația *ExtractMax*). Adăugarea unui nou program în coadă se realizează cu operația *Insert*.

Există diverse modalități de a implementa o coadă cu prioritate: ca un vector, ca o listă înlățuită, ordonată sau nu, etc. Fiecare dintre aceste variante optimizează una dintre cele două operații, celalaltă realizându-se în timp liniar.

O structură de date simplă, ce permite implementarea ambelor operații în timp logaritmic, este *heap*-ul.

## Exerciții propuse

1. Care dintre următoarele situații ar necesita utilizarea unei cozi cu prioritate?
  - a. Simularea activității la un cabinet medical în care medicul tratează pacienții în ordinea sosirii lor.
  - b. Simularea activității la o firmă care trebuie să onoreze cât mai multe contracte. La fiecare moment, firma poate lucra pentru un singur contract. Când se încheie un contract, se va alege contractual cu termenul de predare cel mai apropiat.
  - c. Simularea activității într-o bibliotecă. Cititorii pot restituiri cărți sau solicita cărți spre împrumut. În acest caz, trebuie să căutăm rapid cartea solicitată pentru a verifica dacă este disponibilă.
2. Implementați structura de date abstractă coadă cu prioritate cu ajutorul unui vector sortat, respectiv al unei liste simplu înlățuite. Analizați în aceste cazuri complexitatea funcțiilor *Insert* și *ExtractMax*.
- 3: Să considerăm o coadă cu min-prioritate implementată ca o listă dublu înlățuită ordonată crescător. Care este complexitatea operațiilor de inserare, respectiv de extragere a minimumului?
  - a.  $O(n)$ ,  $O(1)$
  - b.  $O(n)$ ,  $O(n)$
  - c.  $O(\log n)$ ,  $O(1)$
  - d.  $O(n^2)$ ,  $O(1)$

## Min-max heap

Pentru rezolvarea anumitor probleme este necesară uneori determinarea sau extragerea atât a valorilor minime, cât și a valorilor maxime.

O *coadă cu dublă prioritate* este o structură de date abstractă formată din elemente care au asociată o valoare numită cheie sau prioritate și care suportă următoarele operații:

- *Insert(Q, x)*: inserează elementul  $x$  în coadă cu prioritate  $Q$ ;
- *ExtractMax(Q)*: extrage din coadă  $Q$  elementul de valoare maximă;
- *ExtractMin(Q)*: extrage din coadă  $Q$  elementul de valoare minimă.

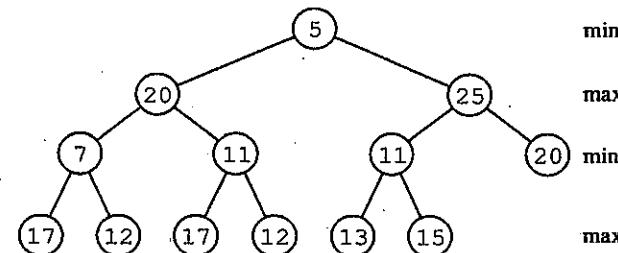
Această structură de date abstractă poate fi implementată eficient utilizând o structură de date numită *min-max heap*.

Un *min-max heap* este un arbore binar complet în care:

- nivelurile sunt, alternativ, niveluri minime, respectiv niveluri maxime, rădăcina aflându-se pe un nivel minim;
- dacă  $x$  este un nod situat pe un nivel minim, valoarea nodului  $x$  este mai mică decât toate valorile din nodurile subarborelui de rădăcină  $x$ , nodul  $x$  numindu-se nod minim;
- dacă  $x$  este un nod situat pe un nivel maxim, valoarea sa este mai mare decât toate valorile din nodurile subarborelui de rădăcină  $x$ , nodul  $x$  numindu-se nod maxim.

Se constată că nivelurile cu număr de ordine par sunt niveluri minime, iar cele cu număr de ordine impar, niveluri maxime.

### Exemplu



Fieind arbori binari compleți, reprezentarea cea mai eficientă a *min-max heap*-urilor este reprezentarea secvențială. Pentru *min-max heap*-ul din figură, reprezentarea secvențială este:

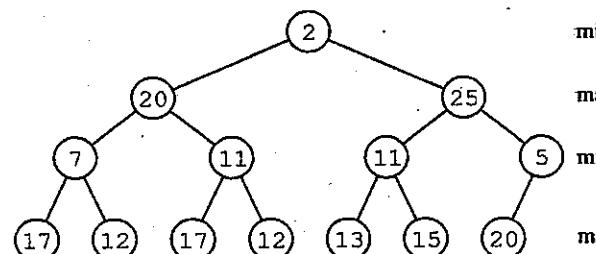
5	20	25	7	11	11	20	17	12	17	12	13	15
1	2	3	4	5	6	7	8	9	10	11	12	13

### Inserarea unui nod într-un min-max heap

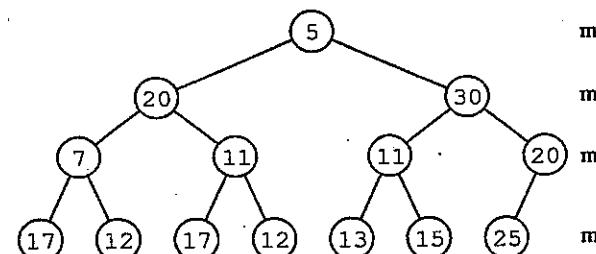
Noul nod se memorează pe prima poziție neocupată din vectorul ce constituie reprezentarea secvențială a *min-max heap*-ului și apoi se restaurează *heap*-ul, „promovând” valoarea nou inserată spre rădăcină, până ajunge în poziția în care structura de *min-max heap* se refac. Evident, la inserare, dimensiunea *min-max heap*-ului crește cu o unitate.

#### Exemplu

Să inserăm nodul cu valoarea 2 în *min-max heap*-ul din exemplul precedent. Noul nod va ocupa poziția 14 în vector, tatăl său fiind nodul de pe poziția 7, cu valoarea 20. Nodul tată este un nod de pe un nivel minim, deci valoarea să fie mai mică decât valorile nodurilor din subarbore. Din acest motiv nodul cu valoarea 2 „promovează” pe nivelul superior, ajungând în acest fel pe un nivel minim, iar valoarea nodului tată coboară în arbore. Nodul nou inserat se află pe poziția 7 pe un nivel minim; continuăm compararea valorii sale cu valorile nodurilor situate pe niveluri minime, „promovând-o” cât timp este necesar, pentru a conserva structura *min-max heap*-ului. Arborele obținut este următorul :



Dacă în arborele din exemplu am inseră un nod cu valoarea 30, pentru că  $30 > 20$  (valoarea tatălui său, situat pe un nivel minim), trebuie să verificăm proprietatea de *min-max heap* relativ la nodurile maxime de pe drumul spre rădăcină. Deci comparăm valoarea nou inserată cu valoarea nodului de pe poziția 3. Cum  $25 < 30$ , valoarea 30 „promovează” în arbore. Continuăm comparațiile cu toate nodurile maxime de pe drumul spre rădăcină, „promovând” noua valoare, până când structura *min-max heap*-ului este restaurată. Rezultă arborele :



```

void Insert(int x)
{
    int niv, fiu, tata;
    H[++n]=x;
    if (n<=1) return;
    fiu=n; //pozitia curenta a valorii x
    niv=log2(n); //nivelul pe care este inserat noul nod
    if (niv%2==0) //nodul este inserat pe un nivel minim
        (tata=n/2; //nodul tata situat pe nivel maxim
        if (x>H[tata])
            //compar cu nodurile maxime de pe drumul spre radacina
            while (tata && x>H[tata])
                { H[fiu]=H[tata];
                  fiu=tata; tata=fiu/4; }
                H[fiu]=x; }
    else //compar cu nodurile minime de pe drumul spre radacina
        (tata=fiu/4;
        while (tata && x<H[tata])
            { H[fiu]=H[tata];
              fiu=tata; tata=fiu/4; }
            H[fiu]=x; }
    else //nodul este inserat pe un nivel maxim
        (tata=n/2; //nodul tata situat pe nivel minim
        if (x<H[tata])
            //compar cu nodurile minime de pe drumul spre radacina
            while (tata && x<H[tata])
                { H[fiu]=H[tata];
                  fiu=tata; tata=fiu/4; }
                H[fiu]=x; }
        else //compar cu nodurile maxime de pe drumul spre radacina
            (tata=fiu/4;
            while (tata && x>H[tata])
                { H[fiu]=H[tata];
                  fiu=tata; tata=fiu/4; }
                H[fiu]=x; }
}
  
```

#### Observație

Complexitatea procedurii de inserare a unei valori într-un *min-max heap* este de  $O(\log n)$ , valoarea noului nod fiind „promovată” în cel mai defavorabil caz până în rădăcina arborelui, un *min-max heap* fiind un arbore binar complet, înălțimea sa este de  $O(\log n)$ .

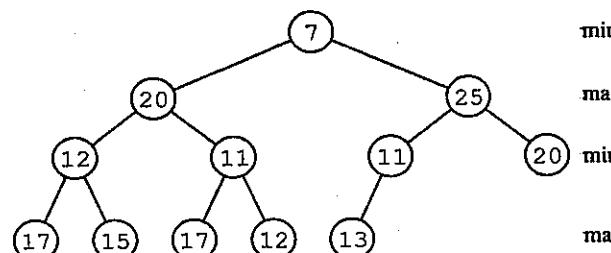
#### Extragerea nodului minim dintr-un min-max heap

Extragerea nodului minim revine la extragerea rădăcinii. Valoarea rădăcinii este înlocuită cu valoarea nodului aflat pe ultima poziție în arbore, apoi aceasta este „retrogradată”, până la restabilirea structurii de *min-max heap*.

Rădăcina se află întotdeauna pe un nivel minim, deci pentru a restaura *min-max heap*-ul, comparăm valoarea plasată în rădăcină cu valoarea celui mai mic dintre nepoți, retrogradând succesiv această valoare până la restabilirea structurii de *min-max heap*. Există situația limită când nivelul pe care se află valoarea este ultimul nivel minim, caz în care valoarea trebuie comparată cu valorile nodurilor (dacă acestea există) de pe nivelul maxim următor și, eventual, schimbate între ele.

### Exemplu

Să considerăm *min-max heap-ul* din exemplul precedent. Valoarea rădăcinii va fi temporar înlocuită cu valoarea 15. Valoarea 15 va fi schimbată cu 7 – valoarea celu mai mic dintre nepoții săi. Valoarea 15 este acum plasată pe ultimul nivel minim și trebuie comparată cu valorile nodurilor de pe nivelul maxim următor. Astfel, va fi schimbată cu valoarea 12. Se obține arborele din figura următoare :



Funcția `ExtractMin(H, n)` extrage elementul minim din *min-max heap*-ul  $H$ , de dimensiune  $n$ :

```

int ExtractMin()
{
    int poz=1, x;
    int gata=0;
    //gata va fi 1 cand terminam comparatiile pe niveluri minime
    int v=H[n--], rez=H[1];
    while (!gata && 4*poz<=n)
        {
            x=MinNepot(poz);
            //x este pozitia celui mai mic nepot al nodului poz
            if (v<H[x])  gata=1;
            else
                {H[poz]=H[x];
                 poz=x; }
        }
    H[poz]=v;
    //compar cu nodurile de pe nivelul maxim urmator
    x=MinFiu(poz); //x pozitia celui mai mic fiu al nodului poz
    if (x<=n)      //exista cel putin un fiu
        if (v>H[x])
            {H[poz]=H[x];
             H[x]=v; }
    return rez;
}

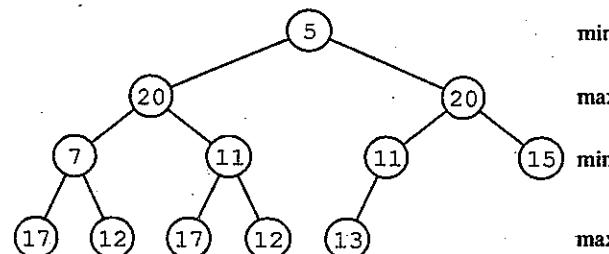
```

### *Extragerea elementului maxim*

Elementul de valoare maximă este fiul rădăcinii cu valoarea cea mai mare. Deci, pentru a extrage maximul dintr-un *min-max heap* se înlocuiește cel mai mare, „dintre” fiilor rădăcinii cu valoarea ultimului nod, „retrogradând” succesiv această valoare până când structura de *min-max heap* este restabilită. În situația-limită când nu există nivel maxim, va fi extrasă valoarea nodului rădăcină.

### *Exemplu*

Să extragem elementul maxim din *min-max heap*-ul din exemplul precedent. Acesta este nodul cu valoarea 25. Valoarea ultimului nod va fi temporar plasată în nodul 3. Nodul extras se află întotdeauna pe un nivel maxim. Vom compara succesiv noua valoare cu valoarea celui mai mare dintre nepoți, eventual retrogradând-o, până la restabilirea structurii de *min-max heap* sau până când nu mai există nepoți. În cazul în care unul dintre fiii său și pe nivelul minim următor ar avea o valoare mai mare le interzicem să schimbăm. Deci interzicem să schimbăm valoarea 15 cu valoarea 20. Obținem arborele:



Funcția `ExtractMax(H, n)` extrage valoarea maximă din *min-max heap*-ul  $H$ , de dimensiune  $n$ :

```

int ExtractMax()
{int poz=2, x, gata=0;
 if (n==1) //exista un singur nod
    {n=0; return H[1];}
 int rez=H[2], v;
 if (n>2 && H[3]>H[2]) {rez=H[3]; poz=3;}
 v=H[n--]; // restauram min-max heap-ul
while (!gata && 4*poz <=n)
    {x=MaxNepot(poz);
     //x este pozitia celui mai mare nepot al nodului poz
     if (v>H[x]) gata=1;
     else {H[poz]=H[x]; poz=x;}}
H[poz]=v;
//compar cu nodurile de pe nivelul minim urmator
x=MaxFiu(poz);
//x este pozitia celui mai mare fiu al nodului poz
if (x <= n) //exista cel putin un fiu
    if (v<H[x]) {H[poz]=H[x]; H[x]=v;}
return rez; }

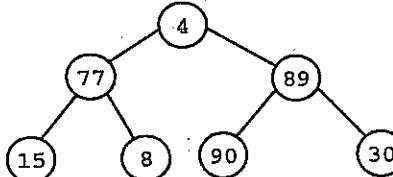
```

*Observație*

Complexitatea operațiilor `ExtractMin()` și `ExtractMax()` este logaritmică, fiind urmată, în cazul cel mai favorabil, de o eventuală actualizare a valorilor nodurilor de pe drumul de la nodul extras până la ultimul nivel.

*Exerciții propuse*

1. Creați un *min-max heap* inserând succesiv valorile: 7, 9, 20, 3, 0, 45, 12, 16, 99, 30, 22, 40, 4, 77, 13, 11, 5, 1.
2. Scrieți o funcție care să verifice dacă un vector specificat ca parametru constituie reprezentarea secvențială a unui *min-max heap*.
3. Extragăți elementul minim, apoi pe cel maxim din *min-max heap*-ul următor:

**3.9. Arbori binari de căutare**

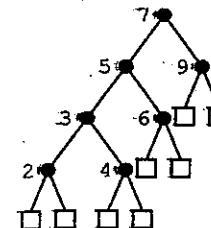
Problema cel mai frecvent întâlnită în practică este, probabil, cea a găsirii rapide a unor informații. De exemplu, căutarea unui cuvânt în dicționar sau a unui număr în carte de telefon. De obicei, informațiile sunt reprezentate sub forma unor structuri, fiecare structură conținând un câmp numit cheie, ce permite identificarea sa în mod unic.

O problemă de căutare constă în identificarea unei structuri cu cheia specificată, în scopul prelucrării informațiilor corespunzătoare. Presupunem că structurile au chei distințe; cazul structurilor având chei identice poate fi tratat în diverse moduri, în funcție de aplicație.

Am studiat deja doi algoritmi de căutare: căutarea secvențială (algoritm liniar) și căutarea binară (algoritm logaritmic). Căutarea binară este un algoritm eficient, dar poate fi aplicată doar în cazul în care mulțimea cheilor este ordonată și nu se modifică (adică nu se efectuează operații de inserare sau stergere a unor valori).

Utilizarea arborilor constituie o metodă simplă de căutare într-o mulțime dinamică și reprezintă unul dintre algoritmii fundamentali în informatică.

Un *arbore binar de căutare* este un arbore binar în care pentru orice nod  $x$  cheile nodurilor din subarborele stâng al lui  $x$  sunt mai mici decât cheia lui  $x$ , iar cheile nodurilor din subarborele drept al lui  $x$  sunt mai mari decât cheia lui  $x$ .

*Exemplu*

Pentru a trata în mod unitar nodurile arborelui am adăugat noduri fictive, notează  $\square$ , care se numesc noduri externe. Astfel, orice căutare fără succes se va termina pe un nod extern.

*Observație*

Prin parcursarea inordine a unui arbore binar de căutare obținem cheile ordonate crescător.

Se numește *dicționar* o structură abstractă de date care suportă trei operații:

1. `Search(x)` : căută cheia  $x$  în structura de date;
2. `Insert(x)` : inserează cheia  $x$  în structură de date dacă este posibil (cheia  $x$  nu se găsește deja în structura de date);
3. `Delete(x)` : șterge cheia  $x$  din structura de date, dacă este posibil (cheia  $x$  se găsește în structura de date).

Arborii binari de căutare pot constitui o implementare eficientă a dicționarelor.

*Căutarea în arbori binari de căutare*

Comparăm valoarea  $x$  cu cheia rădăcinii arborelui :

- dacă informația nodului rădăcină este  $x$ , atunci căutarea se încheie cu succes ;
- dacă informația nodului rădăcină este mai mare decât  $x$ , vom continua căutarea în subarborele stâng, altfel, vom continua căutarea în subarborele drept al rădăcinii.

Să observăm că fiecare cheie din arboarele binar de căutare partajează spațiul de căutare – format din cheile din subarborele corespunzător – în două mulțimi: mulțimea valorilor mai mici decât cheia dată (reprezentată prin subarborele stâng) și mulțimea valorilor mai mari (reprezentată prin subarborele drept).

Funcția următoare căută recursiv cheia  $x$  în arborele cu rădăcina  $r$  și întoarce `NULL` dacă  $x$  nu se află în arbore, respectiv adresa nodului cu cheia  $x$ , în caz contrar.

```

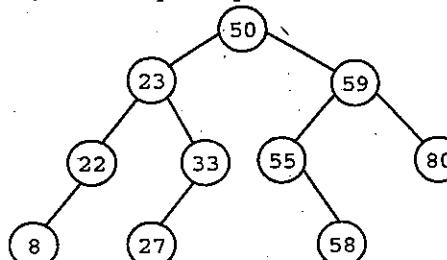
Arbore Cauta(Arbore r, int x)
{
    if (!r) return NULL;
    if (r->inf==x) return r;
    if (x<r->inf) return Cauta(r->st, x);
    return Cauta(r->dr, x);
}
  
```

*Observație*

Analizând complexitatea algoritmului, observăm că pe fiecare nivel în arbore se face cel mult o comparație. În cazul cel mai defavorabil, când nodul căutat se găsește pe ultimul nivel, numărul de comparații este egal cu  $h+1$ , unde  $h$  este înălțimea arborelui. Deci complexitatea algoritmului în cazul cel mai defavorabil este de  $O(h)$ . În cazul unui arbore degenerat, obținem în cazul cel mai defavorabil un algoritm de complexitate  $O(n)$ .

*Exerciții propuse*

- Considerăm următorul arbore binar de căutare. Căutați în acest arbore valoarea 27, apoi valoarea 82, urmărind pas cu pas execuția funcției de căutare.



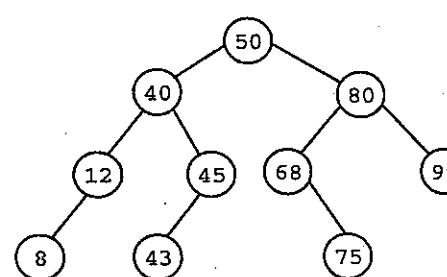
- Scriți o variantă iterativă a funcției de căutare.

*Inserarea unei valori într-un arbore binar de căutare*

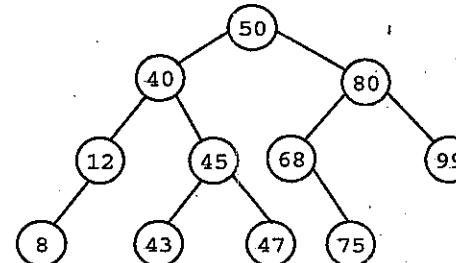
Se dă o cheie  $x$ . Pentru a insera cheia  $x$  într-un arbore binar de căutare efectuăm mai întâi o căutare a valorii  $x$  în arbore. Dacă găsim un nod cu cheia  $x$ , inserarea va fi abandonată, deoarece într-un arbore binar de căutare nu este permisă existența a două noduri cu chei egale. În cazul în care căutarea se încheie fără succes, nodul cu cheia  $x$  va fi inserat ca fiu stâng sau drept al nodului la care s-a terminat căutarea (în funcție de relația dintre  $x$  și cheia nodului respectiv). Prin urmare, procedura de inserare va fi o variantă ușor modificată a funcției de căutare.

*Exemplu*

Să inserăm valoarea 47 în arborele binar de căutare următor:



Se compară 47 cu cheia rădăcinii arborelui (50) și, deoarece 47 este mai mic, ne deplasăm în subarborele stâng. Comparăm acum pe 47 cu rădăcina subarborelui stâng (40). Deoarece 47 este mai mare, ne deplasăm în subarborele drept al nodului cu cheia 40. Comparam pe 47 cu cheia rădăcinii acestui subarbore (45) și deducem că nodul cu valoare 47 trebuie inserat în dreapta nodului cu valoarea 45. Obținem arborele:



```

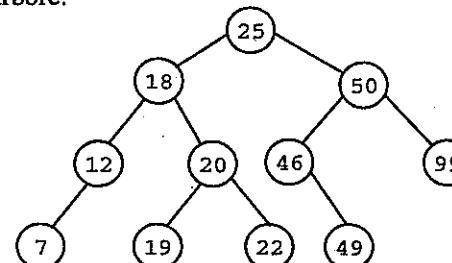
void Inserare(Arbore & r, int x)
{
    Arbore p=new Nod, q=r;
    p->inf=x; p->st=p->dr=NULL; //creez nodul terminal cu cheia x
    if (!r) {r=p; return;} //arborele este vid
    while (1) //caut pozitia valorii x in arbore
        if (q->inf==x) //valoarea x exista deja in arbore
            {delete p; return;}
        if (x<q->inf)
            if (q->st) q=q->st;
            else {q->st=p; return;}
        else
            if (q->dr) q=q->dr;
            else {q->dr=p; return;}
    }
}
  
```

*Observație*

Inserarea unei valori într-un arbore binar de căutare este o variantă ușor modificată a algoritmului de căutare, deci toate observațiile referitoare la complexitate rămân valabile.

*Exerciții propuse*

- Considerăm următorul arbore binar de căutare. Inserați valoarea 23, apoi valoarea 48, în acest arbore.

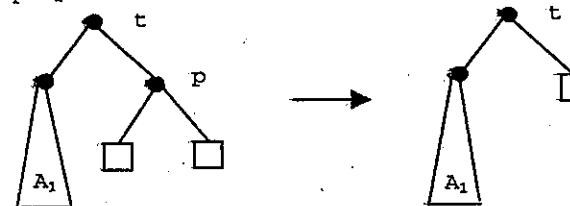


- Scriși o variantă recursivă a funcției de inserare.

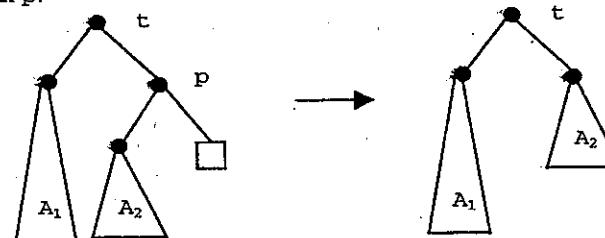
### Ștergerea unui nod de cheie dată dintr-un arbore binar de căutare

Ca și în cazul operației de inserare, ștergerea este precedată de o căutare a nodului cu cheia dată  $x$  în arborele binar. În cazul în care un astfel de nod este găsit – să îl notăm cu  $p$ , iar cu  $t$  să notăm părintele său – există trei situații posibile:

- Nodul  $p$  este un nod terminal. În acest caz, ștergerea este simplă: înlocuim legătura spre  $p$  cu NULL.



- Nodul  $p$  are un singur fiu. Înlocuim legătura spre  $p$  cu o legătură spre unicul fiu al lui  $p$ .

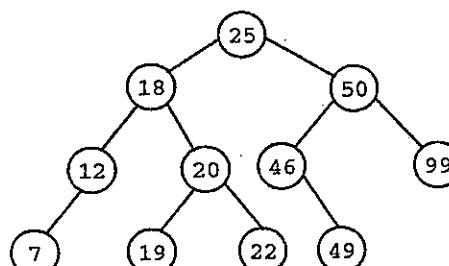


- Nodul  $p$  are exact doi fiți. Vom reduce acest caz la una dintre situațiile a. sau b., conservând proprietatea de arbore binar de căutare. Pentru aceasta vom determina nodul cu cea mai mare cheie din subarborele stâng al lui  $p$ , notat max. Vom înlocui cheia lui  $p$  cu cheia nodului max și vom șterge nodul max, care nu are fiu drept. Deoarece nodul max este situat în subarborele stâng al lui  $p$ , are cheia mai mică decât nodurile din subarborele drept și mai mare decât a nodurilor din subarborele stâng, deci arborele astfel obținut este arbore binar de căutare.

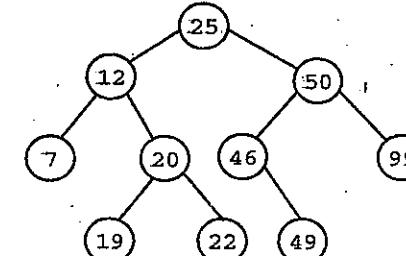
Se poate proceda în mod analog, păstrând consistența arborelui binar de căutare înlocuind cheia lui  $p$  cu cheia nodului cu cea mai mică cheie din subarborele drept al lui  $p$ , notat min, nod care nu are fiu stâng și ștergând apoi nodul min.

#### Exemplu

Să ștergem din arborele binar de căutare următor nodul cu valoarea 18.



Acum nodul cu cheia 18 are doi fiți. Vom determina maximul din subarborele stâng al nodului cu valoarea 18 (nodul cu valoarea 12), înlocuim cheia 18 cu cheia 12, apoi maximul din subarborele stâng (acest nod nu are fiu drept). Obținem arborele:



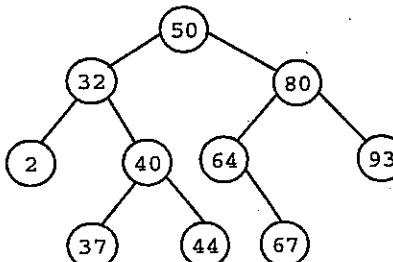
```
void Stergere(Arbore &r, int x)
{
    Arbore p=r, t=NULL, max, tmax, fiu;
    //caut nodul cu cheia x; t indica parintele acestui nod
    while (p && p->inf!=x)
    {
        t=p;
        if (x<p->inf) p=p->st;
        else p=p->dr;
    }
    if (!p) return; //cheia x nu se afla in arbore
    if (p->st && p->dr) //nodul de sters are 2 fiu
    {
        //determin maximumul din subarborele stang
        tmax=p;
        max=p->st;
        while (max->dr)
            {tmax=max;
             max=max->dr; }
        //copiez informatia
        p->inf=max->inf;
        //nodul maxim devine nod de sters
        t=tmax;
        p=max;
    }
    //nodul de sters are cel mult un fiu
    if (p->st) fiu=p->st;
    else fiu=p->dr;
    if (t)
        if (t->st==p) //p este fiu stang al tatalui sau
            t->st=fiu;
        else //p este fiu drept al tatalui sau
            t->dr=fiu;
        else //sterge radacina arborelui
            r=fiu;
        delete p;
    }
}
```

*Observație*

Deoarece ștergerea efectivă a nodului se face în timp constant, complexitatea algoritmului de ștergere se reduce la complexitatea algoritmului de căutare a cheii în arbore. În concluzie, timpul de execuție a operațiilor de căutare, inserare, ștergere depinde de forma arborelui de căutare, fiind în cel mai defavorabil caz de  $O(h)$ , unde  $h$  este înălțimea arborelui. În cazul în care arborele este degenerat, complexitatea algoritmilor de căutare, inserare și ștergere devine liniară. În cazul în care arborele este echilibrat, înălțimea lui este aproximativ  $\log n$ , prin urmare operațiile devin eficiente.

*Exerciții propuse*

1. Considerăm următorul arbore binar de căutare. Stergeți, în ordine, din arbore nodurile care conțin valorile 50, 80, 40, 2:



2. Scrieți o funcție care să determine nodul de cheie minimă dintr-un arbore binar de căutare nevid specificat ca parametru.
3. Scrieți o funcție care să primească drept parametru adresa rădăcinii unui arbore binar de căutare și adresa unui nod oarecare al arborelui și care să returneze ca rezultat adresa succesorului inordine al nodului dat. Numim succesorul inordine al unui nod dat  $x$ , nodul cu cea mai mică cheie mai mare decât cheia lui  $x$ . Altfel spus, succesorul inordine al unui nod este nodul care urmează nodului respectiv în parcursarea în inordine a arborelui binar de căutare.
4. Scrieți o funcție care să primească drept parametru adresa rădăcinii unui arbore binar de căutare și adresa unui nod oarecare al arborelui și care să returneze ca rezultat adresa predecesorului inordine al nodului dat. Numim predecesorul inordine al unui nod dat  $x$ , nodul cu cea mai mare cheie mai mică decât cheia lui  $x$ . Altfel spus, predecesorul inordine al unui nod este nodul care precedă nodul specificat în parcursarea inordine a arborelui.

*Arbore binari de căutare optimali*

Fie  $a_1 < a_2 < \dots < a_n$  o mulțime statică de chei (adică nu se execută operații de inserare sau ștergere de chei din mulțime).

Să notăm cu  $p_i$  probabilitatea de a căuta cheia  $a_i$ , pentru orice  $i=1, 2, \dots, n$ . De asemenea, să notăm cu  $q_i$  probabilitatea de a căuta o cheie din intervalul  $(a_i, a_{i+1})$ , pentru orice  $i=\{0, 1, 2, \dots, n\}$  (considerăm că  $a_0=-\infty$  și  $a_{n+1}=+\infty$ ).

Costul unei căutări cu succes este egal cu numărul de comparații care se execută pentru a identifica nodul care conține cheia căutată înmulțit cu probabilitatea de a căuta cheia respectivă. În total, costul căutărilor cu succes este  $p_1 * (\text{nivel}(a_1)+1) + p_2 * (\text{nivel}(a_2)+1) + \dots + p_n * (\text{nivel}(a_n)+1)$ .

O căutare fără succes se termină într-un nod extern, nod corespunzător unui interval de forma  $(a_i, a_{i+1})$ . Costul unei căutări fără succes este egal cu numărul de comparații efectuate până la identificarea nodului extern corespunzător valorii căutate, înmulțit cu probabilitatea corespunzătoare intervalului în care se află valoarea respectivă. În total, costul căutărilor fără succes va fi:

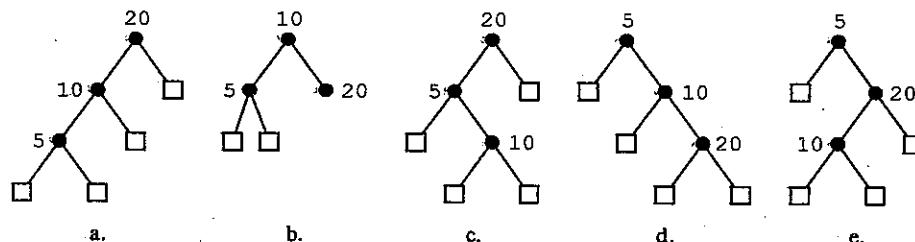
$$q_0 * \text{nivel}(\text{extern}_0) + q_1 * \text{nivel}(\text{extern}_1) + \dots + q_n * \text{nivel}(\text{extern}_n).$$

Costul unui arbore binar de căutare este egal cu suma dintre costurile căutărilor cu succes și costurile căutărilor fără succes.

Numim *arbore binar de căutare optimal* pentru mulțimea  $a_1 < a_2 < \dots < a_n$  un arbore binar de căutare de cost minim.

*Exemplu*

Să considerăm  $a_1=5$ ,  $a_2=10$ ,  $a_3=20$ . Arborii binari de căutare pentru această mulțime de valori sunt:



Dacă vom considera  $p_1=p_2=p_3=1/7$  și  $q_0=q_1=q_2=q_3=1/7$  (deci orice valoare este căutată în arbore cu aceeași probabilitate) obținem:

$$\begin{aligned} \text{cost(arbore a)} &= 15/7 \\ \text{cost(arbore b)} &= 13/7 \\ \text{cost(arbore c)} &= 15/7 \\ \text{cost(arbore d)} &= 15/7 \\ \text{cost(arbore e)} &= 15/7 \end{aligned}$$

Așa cum era de așteptat, arborele binar de căutare optimal este b.

Dar, pentru  $p_1=0.5$ ;  $p_2=0.1$ ;  $p_3=0.05$  și  $q_0=0.15$ ;  $q_1=0.1$ ;  $q_2=0.05$ ;  $q_3=0.05$  obținem:

$$\begin{aligned} \text{cost(arbore a)} &= 2.65 \\ \text{cost(arbore b)} &= 1.9 \\ \text{cost(arbore c)} &= 2.05 \\ \text{cost(arbore d)} &= 1.5 \\ \text{cost(arbore e)} &= 1.6 \end{aligned}$$

Deci arborele binar de căutare optimal este arborele d.

Pentru determinarea arborelui binar de căutare optimal pentru o mulțime dată de valori  $a_1 < a_2 < \dots < a_n$ , o primă idee ar fi să generăm toți arborii binari de căutare pentru mulțimea de valori dată, să calculăm costul fiecărui arbore și să selectăm arborele de cost minim.

Dar numărul arborilor binari cu  $n$  vârfuri date este:

$$B(n) = O\left(\frac{4^n}{n^{3/2}}\right)$$

ceea ce face ca aplicarea metodei să nu fie posibilă pentru valori mari ale lui  $n$ .

Făcând unele observații referitoare la proprietățile arborilor binari de căutare optimali, vom descrie un algoritm mult mai eficient, bazat pe metoda programării dinamice.

Să notăm pentru orice  $\forall 1 \leq i < j \leq n$ :

$T_{i,j}$  = arborele binar de căutare optimal pentru  $a_{i+1} \dots a_j$ ;  $T_{i,i}$  este arborele vid;

$C_{i,j}$  = costul arborelui  $T_{i,j}$ ;  $C_{i,i}=0$ ;

$r_{i,j}$  = rădăcina arborelui  $T_{i,j}$ ;

$w_{i,j}=q_1+q_{i+1}+p_{i+1}+q_{i+2}+p_{i+2}+\dots+q_j+p_j$ ;  $w_{i,i}=q_i$ .

Conform acestor notații:

$T_{0,n}$  este arborele binar de căutare optimal pentru  $a_1, \dots, a_n$ ;

$C_{0,n}$  este costul arborelui binar de căutare optimal  $T_{0,n}$ ;

$r_{0,n}$  este rădăcina arborelui binar de căutare optimal  $T_{0,n}$ ;

$w_{0,n}=1$ .

Fie  $T_{i,j}$  un arbore binar de căutare optimal pentru  $a_{i+1}, \dots, a_j$  și  $r_{i,j}=k$  ( $i < k \leq j$ ).

$T_{i,j}$  are doi subarbore: S subarborele stâng, care conține cheile  $a_{i+1}, \dots, a_{k-1}$ , și D subarborele drept, care conține cheile  $a_{k+1}, \dots, a_j$ . Atunci:

$$C_{i,j}=p_k+w_{i,k-1}+w_{k,j}+\text{cost}(S)+\text{cost}(D)=w_{i,j}+\text{cost}(S)+\text{cost}(D).$$

Din această relație deducem că  $T_{i,j}$  este optimal dacă și numai dacă S și D sunt optimali, deci  $S=T_{i,k-1}$  și  $D=T_{k,j}$  (altfel înlocuind S sau D cu un arbore de cost mai mic, am obține un arbore pentru  $a_{i+1}, \dots, a_j$  de cost mai mic decât  $C_{i,j}$ , în contradicție cu ipoteza că  $T_{i,j}$  este optimal).

Problema are prin urmare substructură optimală pe care o caracterizăm prin următoarea relație de recurență:

$$C_{i,j}=C_{i,k-1}+C_{k,j}+w_{i,j}$$

Cum  $T_{i,j}$  este optimal, rădăcina sa  $r_{i,j}=k$  trebuie să fie selectată astfel încât să se obțină un cost minim, deci:

$$C_{i,j}=w_{i,j}+\min\{C_{i,k-1}+C_{k,j} \mid i < k \leq j\}$$

Vom rezolva această relație de recurență în mod bottom-up.

Construirea unui arbore binar de căutare optimal prin metoda programării dinamice are complexitatea timp de  $O(n^3)$ . Se poate îmbunătăți timpul de execuție al algoritmului, utilizând un rezultat datorat lui D.E. Knuth, conform căruia putem restrânge domeniul în care se caută valoarea ce minimizează expresia lui  $C_{i,j}$  de la intervalul  $[i+1, j]$  la intervalul  $[r_{i,j-1}, r_{i+1,j}]$ . În acest caz procedura de calcul va fi  $O(n^2)$ .

```

int n, a[NMax];
double p[NMax], q[NMax];
double c[NMax][NMax], w[NMax][NMax];
int r[NMax][NMax];
void initializare()
{
    int i;
    ifstream fin("abc.in");
    fin>>n;
    for (i=1; i<=n; i++) fin>>a[i]>>p[i];
    for (i=0; i<=n; i++) fin>>q[i];
    for (i=0; i<n; i++)
        { r[i][i]=0; c[i][i]=0; w[i][i]=q[i];
          w[i][i+1]=q[i]+q[i+1]+p[i+1];
          c[i][i+1]=w[i][i+1]; r[i][i+1]=i+1;
        }
    w[n][n]=q[n]; c[n][n]=0; r[n][n]=0;
}

int knuthmin(int i, int j)
{
    int l, k;
    k=r[i][j-1];
    for (l=r[i][j-1]+1; l<=r[i+1][j]; l++)
        if (c[i][k-1]+c[k][j]>c[i][l-1]+c[l][j]) k=l;
    return k;
}

void calcul()
{
    int d, i, k, j;
    for (d=2; d<=n; d++)
        for (i=0; i<=n-d; i++)
            for (j=i+d;
                 w[i][j]=w[i][j-1]+q[j]+p[j];
                 k=knuthmin(i, j);
                 c[i][j]=w[i][j]+c[i][k-1]+c[k][j];
                 r[i][j]=k;
            )
}
}

```

Vom face crearea arborelui de căutare optimal în mod recursiv:

```

Arbore constrarb(int i, int j)
{
    Arbore aux;
    if (!r[i][j]) return NULL;
    aux= new Nod; aux->inf=r[i][j];
    aux->st=constrarb(i, r[i][j]-1);
    aux->dr=constrarb(r[i][j], j);
    return aux;
}

```

### Exercițiu propus

1. Construji un arbore de căutare optimal pentru mulțimea de valori  $a=(5, 10, 15, 20, 25)$ , cu  $p=(1/20, 2/20, 2/20, 3/20, 4/20)$  și  $q=(4/20, 2/20, 2/20, 1/20, 1/20)$ .

### 3.10. Reprezentarea mulțimilor disjuncte

Dată fiind o mulțime finită  $U$  și  $S = \{S_1, S_2, \dots, S_k\}$  o partitie a mulțimii  $U$ , problema constă în a proiecta o structură de date care să permită executarea eficientă a următoarelor două operații fundamentale:

- $\text{Find}(x)$ : determină mulțimea  $S_i \in S$  căreia îi aparține elementul  $x$ ,  $x \in U$ ;
- $\text{Union}(S_i, S_j)$ : unește mulțimea  $S_i$  cu mulțimea  $S_j$ , obținându-se un nou element al mulțimii  $S$ , ce va înlocui  $S_i$  și  $S_j$ .

#### Observații

1. Această problemă este cunoscută și sub denumirea de *problema claselor de echivalență* ( $x, y$  se vor numi echivalente dacă aparțin aceleiași mulțimi  $S_i \in S$ , sau, altfel spus,  $\text{Find}(x) = \text{Find}(y)$ ), concepție de relație de echivalență și partitie reprezentând două abordări diferite ale aceleiași structuri matematice.
2. Dat fiind un graf neorientat, componentele conexe ale grafului constituie o partitie a mulțimii vârfurilor grafului. Un algoritm de determinare a componentelor conexe ale unui graf neorientat poate fi formulat cu ajutorul operațiilor  $\text{Union}$  -  $\text{Find}$  astfel:

Pentru  $\forall i \in \{1, 2, \dots, n\}$ ,  $S_i = \{i\}$ ;

Pentru  $\forall [i, j]$  muchie în graf

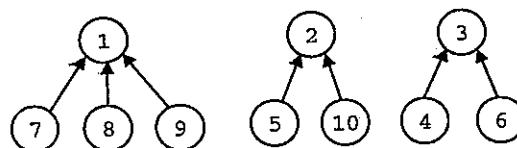
$A = \text{Find}(i)$ ;  $B = \text{Find}(j)$ ;

Dacă  $A \neq B$  atunci  $\text{Union}(A, B)$ ;

O soluție eficientă este reprezentarea mulțimilor disjuncte cu ajutorul arborilor cu rădăcină. Fiecare arbore reprezintă o mulțime, iar fiecare nod din arbore un element al mulțimii. Rădăcina arborelui va fi elementul reprezentativ al mulțimii.

#### Exemplu

Să considerăm  $n=10$  și o partitie a mulțimii  $\{1, 2, \dots, 9, 10\}$  formată din  $S_1 = \{1, 7, 8, 9\}$ ,  $S_2 = \{2, 5, 10\}$ ,  $S_3 = \{3, 4, 6\}$ . Partitia  $S = \{S_1, S_2, S_3\}$  poate fi reprezentată ca o pădure formată din trei arbori:



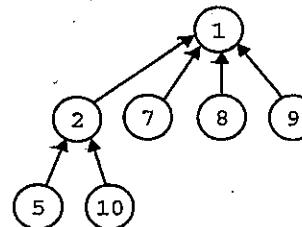
Reprezentăm arborii prin referințe ascendente, deci pentru fiecare nod din arbore, cu excepția rădăcinii, reținem legătura spre părintele său:  $\text{tata}[x] = \text{nodul părinte al lui } x$  sau  $0$ , dacă  $x$  este rădăcina arborelui.

Operația  $\text{Find}(x)$  constă în a determina rădăcina arborelui al cărui nod este  $x$ .

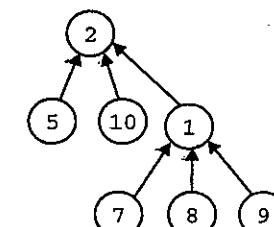
```

int Find(int x)
{
    while (tata[x])
        x=tata[x];
    return x;
}
  
```

Operația  $\text{Union}(i, j)$  se poate realiza transformând unul dintre arbori în subarborele celuilalt. Reuniunea  $S_1 \cup S_2$  poate fi reprezentată în două moduri:



sau



Rădăcina unuia dintre arbori devine părintele rădăcinii celuilalt arbore.

```

void Union(int i, int j)
{
    tata[i]=j;
}
  
```

#### Exemplu

Să considerăm partitia  $S = \{(1), (2), \dots, (n)\}$ . Configurația inițială constă dintr-o pădure, în care fiecare arbore este format doar din rădăcină:  $\text{tata}[i] = 0$ ,  $\forall i \in \{1, 2, \dots, n\}$ . Să considerăm acum următoarea secvență de operații  $\text{Union}$  și  $\text{Find}$ :  $\text{Union}(1, 2)$ ,  $\text{Find}(1)$ ,  $\text{Union}(2, 3)$ ,  $\text{Find}(1)$ ,  $\text{Union}(3, 4)$ ,  $\text{Find}(1)$ , ...,  $\text{Find}(1)$ ,  $\text{Union}(n-1, n)$ . Această secvență conduce la un arbore degenerat. Cum timpul necesar operației  $\text{Union}$  este constant, cele  $n-1$  operații  $\text{Union}$  au timpul de execuție de  $O(n)$ . Dar fiecare operație  $\text{Find}(1)$  parcurge tot drumul de la nodul 1 până la rădăcină. Cum timpul de execuție necesar operației  $\text{Find}$  pentru un nod de pe nivelul  $i$  este  $O(i)$ , obținem un timp de execuție de  $O(n^2)$  pentru cele  $n-2$  operații  $\text{Find}(1)$ .

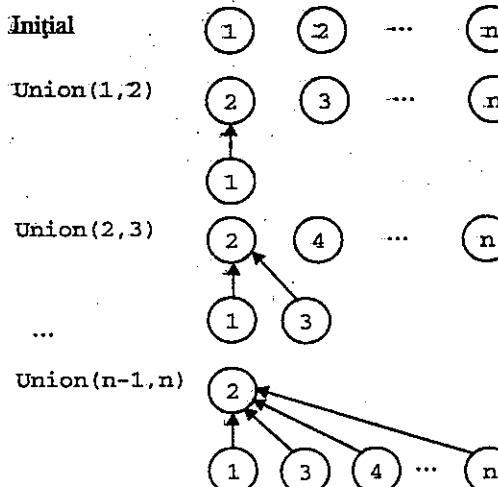
Putem îmbunătăți eficiența operațiilor  $\text{Union}$  și  $\text{Find}$ , aplicând o regulă de ponderare pentru operațiile  $\text{Union}$ .

#### Regula de ponderare

Dacă numărul de niveluri din arboarele cu rădăcina  $i$  este mai mic decât numărul de niveluri din arboarele cu rădăcina  $j$ , atunci  $j$  va deveni părintele lui  $i$ ; altfel,  $i$  va deveni părintele lui  $j$ .

**Exemplu**

Utilizând regula de ponderare, secvența de operații Union și Find din exemplul precedent va conduce la următorii arbori:



În acest caz, timpul de execuție necesar operațiilor Find este  $O(n)$ , deoarece arborii obținuți au înălțimea cel mult egală cu 1. Totuși, există și cazuri defavorabile.

**Lemă**

Dacă A este un arbore cu  $n$  noduri obținut în urma aplicării operației Union folosind regula de ponderare, înălțimea arborelui este cel mult egală cu  $\lfloor \log n \rfloor$ .

**Demonstrație**

Vom proceda prin inducție după  $n$ , numărul de noduri. Pentru  $n=1$ , rezultatul este evident. Presupunem că afirmația este adeverată pentru arbori cu  $i$  noduri ( $1 \leq i \leq n-1$ ), obținuți în urma aplicării algoritmului Union. Să demonstrăm că înălțimea oricărui arbore cu  $n$  noduri  $A_n$ , obținut în urma aplicării algoritmului Union este cel mult egală cu  $\lfloor \log n \rfloor$ .

Fie  $\text{Union}(j,k)$  ultima operație Union executată pentru obținerea arborelui  $A_n$ . Notăm cu  $m$  numărul de noduri din arborele cu rădăcina  $j$ . Arborele cu rădăcina  $k$  are  $m-n$  noduri. Putem presupune, fără a restrângere generalitatea, că  $1 \leq m \leq n/2$ . Deci înălțimea  $h$  a arborelui  $A_n$  va fi egală cu înălțimea arborelui cu rădăcina  $k$  sau cu o unitate mai mare decât înălțimea arborelui cu rădăcina  $j$ .

În primul caz:  $h \leq \lfloor \log(m-n) \rfloor \leq \lfloor \log n \rfloor$ .

În cel de-al doilea caz:  $h \leq \lfloor \log m \rfloor \leq \lfloor \log(n/2) \rfloor + 1 \leq \lfloor \log n \rfloor$ .

Pentru a aplica regula de ponderare este necesar ca pentru fiecare arbore să cunoască numărul de niveluri. Fie  $h$  un vector, în care  $h[i]$  reprezintă numărul de niveluri din arborele cu rădăcina  $i$ . Acest vector va fi actualizat eventual la operații Union.

```
void Union_ponderare(int x, int y)
{
    if (h[x] > h[y]) tata[y]=x;
    else
        {tata[x]=y;
        if (h[x]==h[y]) h[y]++;
        }
}
```

Lema 1 garantează un timp de execuție de  $O(\log n)$  pentru operația  $\text{Find}(x)$ , pentru orice nod  $x$  dintr-un arbore obținut prin operații Union folosind regula de ponderare.

Pentru a îmbunătăți în continuare timpul de execuție a operației  $\text{Find}(x)$  vom utiliza o regulă de compresie.

**Regula de compresie**

Orice nod  $y$  de pe drumul de la rădăcina arborelui la nodul  $x$  va deveni fiu al rădăcinii arborelui.

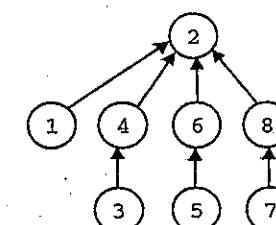
```
int Find_compresie(int x)
{
    int r=x;
    //determinam radacina arborelui
    while (tata[r]) r=tata[r];
    //comprimam drumul de la x la radacina
    int y=x;
    while (y!=r)
        {t=tata[y];
        tata[y]=r;
        y=t;}
    return r;
}
```

Funcția  $\text{Find}_\text{compresie}()$  parurge drumul de la nodul  $x$  la rădăcină de două ori, o dată pentru determinarea rădăcinii, a doua oară pentru comprimarea drumului de la  $x$  la rădăcină. Totuși, această modificare reprezintă o îmbunătățire, deoarece într-o secvență de operații Union-Find, timpul total de execuție va fi mai mic.

**Exemplu**

Să considerăm arborele obținut prin secvența de operații Union din exemplul precedent. Executând prima oară operația  $\text{Find}(8)$  obținem arborele următor.

Au fost necesare trei deplasări pe legături ascendente pentru a identifica rădăcina arborelui și apoi, pentru compresia drumului de la rădăcină la nodul 8, alte două deplasări. Dar următoarele apeluri ale funcției  $\text{Find}(8)$  vor necesita o singură deplasare până la rădăcina arborelui. Astfel costul total al unei secvențe de operații  $\text{Find}(8)$  va fi mai mic.



### Exerciții propuse

- Se consideră  $n=10$  și partitia  $S_i = \{i\}$  pentru orice  $i=1, \dots, 10$ . Să se ilustreze pas cu pas efectul următoarelor operații: Union(2,5), Union(2,8), Union(1,7), Union(8,9), Union(7,3), Union(2,10), Find(10), Find(5), Find(7), Union(4,10), Find(2), Find(4), Find(10). Efectuați apoi același exercițiu utilizând regula de ponderare și regula de compresie. Comparați cele două variante din punctul de vedere al eficienței.
- Implementați algoritmul de descompunere în componente conexe a unui graf neorientat, utilizând operațiile Union și Find.

### 3.11. Arbori de compresie Huffman

În analiza algoritmilor pe care i-am studiat până acum, prioritățea a fost complexitatea timp. Acum ne punem problema elaborării unor algoritmi care să micșoreze spațiul necesar memorării unui fișier.

Tehnicile de compresie sunt utile pentru fișiere text, în care anumite caractere apar cu o frecvență mai mare decât alele, pentru fișiere ce codifică imagini sau sunt reprezentări digitale ale sunetelor ori ale unor semnale analogice, care pot conține numeroase motive ce se repetă. Chiar dacă astăzi capacitatea dispozitivelor de memorare a crescut foarte mult, algoritmi de compresie a fișierelor rămân foarte importanți, datorită volumului tot mai mare de informații ce trebuie stocate. În plus, compresia este deosebit de utilă în comunicații, transmiterea informațiilor fiind mult mai costisitoare decât prelucrarea lor.

Una dintre cele mai răspândite tehnici de compresie a fișierelor text, care, în funcție de caracteristicile fișierului ce urmează a fi comprimat, conduce la reducerea spațiului de memorie necesar cu 20%-90%, a fost descoperită de D. Huffman în 1952 și poartă numele de *codificare (cod) Huffman*. În locul utilizării unui cod în care fiecare caracter să fie reprezentat pe 8 biți (lungime fixă), se utilizează un cod mai scurt pentru caracterele care sunt mai frecvente și coduri mai lungi pentru cele care apar mai rar.

Să presupunem că avem un fișier de 100000 de caractere din alfabetul  $\{a, b, c, d, e, f\}$  pe care dorim să-l memorăm cât mai compact. Dacă am folosi un cod de lungime fixă, pentru cele 6 caractere, ar fi necesari câte 3 biți. De exemplu, pentru codul următor ar fi necesari în total 300000 biți.

	a	b	c	d	e	f
cod fix	000	001	010	011	100	101

Să presupunem acum că frecvențele cu care apar în text cele 6 caractere sunt:

	a	b	c	d	e	f
frecvență	45	13	12	16	9	5

Considerând următorul cod de lungime variabilă ar fi necesari doar 224000 biți (deci o reducere a spațiului de memorie cu aproximativ 25%).

cod variabil	a	b	c	d	e	f
0		101	100	111	1101	1100

Problema se reduce deci la a asocia fiecărui caracter un cod binar, în funcție de frecvență, astfel încât să fie posibilă decodificarea fișierului comprimat, fără ambiguități. De exemplu, dacă am fi codificat a cu 1001 și b cu 100101, când citim în fișierul comprimat secvența 1001 nu putem decide dacă este vorba de caracterul a sau de o parte a codului caracterului b.

Ideeasă de a folosi separatori între codurile caracterelor pentru a nu crea ambiguități ar conduce la mărirea dimensiunii codificării.

Pentru a evita ambiguitățile este necesar ca nici un cod de caracter să nu fie prefix al unui cod asociat unui caracter (un astfel de cod se numește *cod prefix*).

#### Codul Huffman

D. Huffman a elaborat un algoritm *Greedy* care construiește un cod prefix optim, denumit cod Huffman. Prima etapă în construcția codului Huffman este calcularea numărului de apariții ale fiecărui caracter în text. Există situații în care putem utiliza frecvențele standard de apariție a caracterelor, calculate în funcție de limbă sau de specificul textului.

Fie  $C = \{c_1, c_2, \dots, c_n\}$  mulțimea caracterelor dintr-un text, respectiv  $f_1, f_2, \dots, f_n$ , numărul lor de apariții. Dacă  $l_i$  ar fi lungimea sirului ce codifică simbolul  $c_i$ , atunci lungimea totală a reprezentării ar fi:

$$L = \sum_{i=1}^n l_i \cdot f_i$$

Scopul nostru este de a construi un cod prefix care să minimizeze această expresie. Pentru aceasta construim un arbore binar în manieră *bottom-up* astfel:

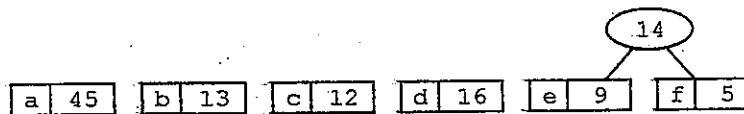
- Initial, considerăm o partitură a mulțimii  $C = \{ (c_1, f_1), (c_2, f_2), \dots, (c_n, f_n) \}$ , reprezentată printr-o pădure de arbori formați dintr-un singur nod.
- Pentru a obține arboarele final, se execută  $n-1$  operații de unificare. Unificarea a doi arbori  $A_1$  și  $A_2$  constă în obținerea unui arbore  $A$ , al cărui subarbore stâng este  $A_1$ , subarbore drept  $A_2$ , iar frecvența rădăcinii lui  $A$  este suma frecvențelor rădăcinilor celor doi arbori. La fiecare pas unificăm doi arbori ale căror rădăcini au frecvențe cele mai mici.

#### Exemplu

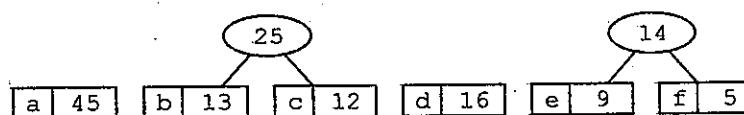
Arboarele Huffman asociat caracterelor  $\{a, b, c, d, e, f\}$  cu frecvențele  $\{45, 13, 12, 16, 9, 5\}$  se construiește pornind de la cele cinci noduri din figură:

a   45	b   13	c   12	d   16	e   9	f   5
--------	--------	--------	--------	-------	-------

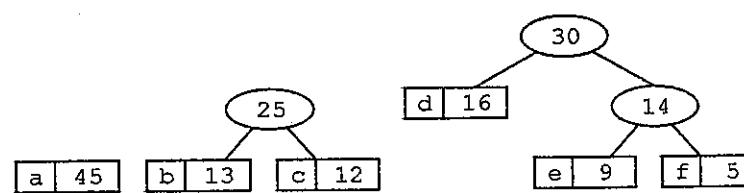
Pas 1 – unific arborii corespunzători lui e și f, deoarece au frecvențele cele mai mici:



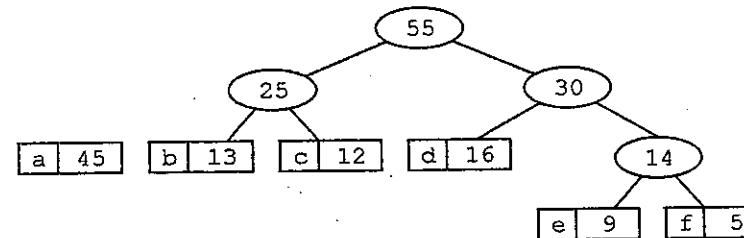
Pas 2 – unific arborii corespunzători lui b și c:



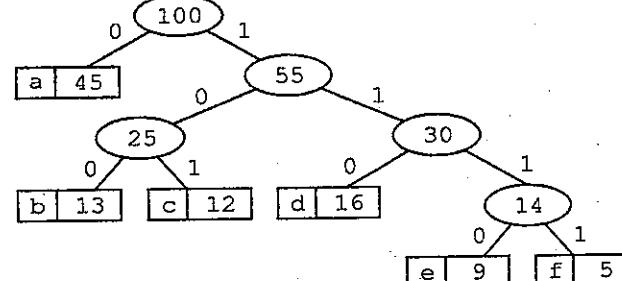
Pas 3 – unific arborele corespunzător lui d și arborele obținut la primul pas, cu rădăcina ce are frecvența 14:



Pas 4 – unific arborii cu rădăcinile care au frecvențele 25 și 30.



Pas 5 – unificând ultimii doi arbori obțin arborele Huffman asociat acestui set de caractere cu frecvențele specificate inițial.



Nodurile terminale vor conține un caracter și frecvența corespunzătoare caracterului; nodurile interioare conțin suma frecvențelor caracterelor corespunzătoare nodurilor terminale din subarbori.

Arborele Huffman obținut va permite asocierea unei codificări binare fiecărui caracter. Caracterele fiind frunze în arborele obținut, se va asocia pentru fiecare deplasare la stânga pe drumul de la rădăcină la nodul terminal corespunzător caracterului un 0, iar pentru fiecare deplasare la dreapta un 1.

Obținem codurile :

cod	a	b	c	d	e	f
	0	100	101	110	1110	1111

Observații

- Caracterele care apar cel mai frecvent sunt mai aproape de rădăcină și astfel lungimea codificării va avea un număr mai mic de biți.
- La fiecare pas am selectat cele mai mici două frecvențe pentru a unifica arborii corespunzători. Pentru extragerea eficientă a minimului vom folosi un *min-heap*. Astfel timpul de execuție pentru operațiile de extragere minim, inserare și stergere va fi, în cazul cel mai defavorabil,  $O(\log n)$ .

### Algoritm de construcție a arborelui Huffman

Pas 1. Inițializare :

- fiecare caracter reprezintă un arbore format dintr-un singur nod ;
- organizăm caracterele ca un *min-heap*, în funcție de frecvențele de apariție.

Pas 2. Se repetă de  $n-1$  ori :

- extrage succesiv X și Y, două elemente din *heap* ;
- unifică arborii X și Y (crează Z, un nou nod ce va fi rădăcina arborelui ;  $Z \rightarrow st = X$ ;  $Z \rightarrow dr = Y$ ;  $Z \rightarrow freqv = X \rightarrow freqv + Y \rightarrow freqv$ );
- insereză Z în *heap*.

Pas 3. Singurul nod rămas în *heap* este rădăcina arborelui Huffman. Se generează codurile caracterelor parcurgând arborele Huffman.

### Analiza complexității

Inițializarea *heap*-ului este liniară. Pasul 2 se repetă de  $n-1$  ori și presupune două operații de extragere a minimului dintr-un *heap* și o operație de inserare a unui element în *heap*, care au timpul de execuție de  $O(\log n)$ . Deci complexitatea algoritmului de construcție a arborelui Huffman este de  $O(n \log n)$ .

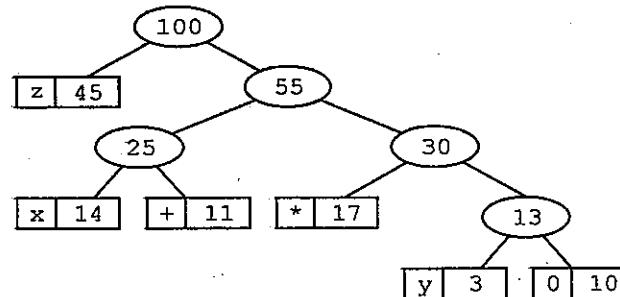
### Exerciții propuse

- Să considerăm un text de 1000000 de caractere din alfabetul {a, b, c, d, e, f, g, h, i, k}. Frecvențele de apariție ale caracterelor în text sunt :

Caracter	a	b	c	d	e	f	g	h	i	k
Frecvență	15%	3%	20%	4%	25%	15%	5%	2%	10%	1%

Construji un cod Huffman și estimați eficiența utilizării sale pentru acest text.

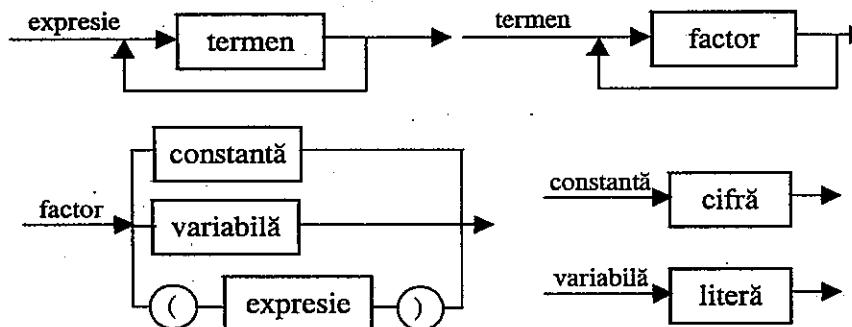
2. Să considerăm următorul arbore Huffman. Construiți codul fiecărui caracter.



3. Scrieți un program care să citească un text, să determine numărul de apariții ale fiecărui caracter în text, să construiască un cod Huffman optimal pentru textul dat, apoi să codifice textul utilizând codul Huffman construit.

### 3.12. Arbori asociați expresiilor aritmetice

Numim *expresie aritmetică* o construcție definită prin următoarele diagrame de sintaxă:



#### Observație

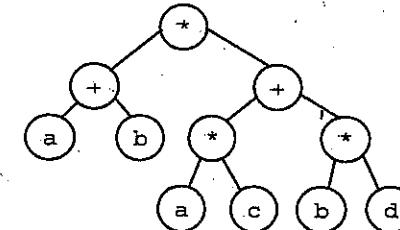
Am ales, pentru simplitate, expresii aritmetice pentru care operanții sunt formati dintr-un singur caracter, iar operatorii pot fi doar + și \*. De asemenea, nu se acceptă spații. Aceste restricții nu restrâng generalitatea.

Oricarei expresii aritmetice i se poate asocia un arbore binar strict, construit după următoarele reguli :

- dacă expresia este formată dintr-un singur operand, arborele binar asociat este constituit dintr-un singur nod care conține drept informație operandul respectiv;
- dacă expresia este de forma  $E = E_1 \text{ op } E_2$ , unde op este un operator, iar  $E_1$  și  $E_2$  sunt expresii aritmetice, arborele binar corespunzător conține în rădăcină operatorul, subarborele stâng fiind arborele binar asociat expresiei  $E_1$ , iar subarborele drept arborele binar asociat expresiei  $E_2$ .

#### Exemplu

Arborele binar asociat expresiei  $E = (a+b)*(a*c+b*d)$  este :



#### Observații

1. Într-un arbore binar asociat unei expresii aritmetice nodurile terminale conțin operanți, iar nodurile interioare conțin operatori.
2. Operatorii utilizati în cadrul expresiei fiind binari, arborele binar asociat expresiei aritmetice este strict.

Parcurgând în preordine arborele binar asociat expresiei aritmetice obținem notația poloneză a expresiei (forma prefixată).

#### Exemplu

Pentru expresia aritmetică  $E = (a+b)*(a*c+b*d)$ , notația poloneză este  $+ab+*ac*bd$ .

Această notație a fost introdusă de matematicianul polonez J. Lucasiewicz și permite scrierea fără paranteze a unei expresii aritmetice.

#### Implementare

Vom defini trei funcții, câte una pentru fiecare unitate sintactică: factor, termen, expresie. Funcția Factor(i) creează arborele corespunzător factorului care începe la poziția i (dacă pe poziția i se află o paranteză deschisă, deducem că factorul este constituit dintr-o expresie cuprinsă între paranteze rotunde, deci pentru a crea arborele vom apela recursiv indirect funcția CreareArbore()); dacă pe poziția i se află un operand, arborele corespunzător acestui factor este constituit dintr-un singur nod terminal).

Funcția Termen(i) creează arborele corespunzător termenului de pe poziția i. Un termen este constituit dintr-un singur factor (în acest caz arborele corespunzător termenului coincide cu arborele corespunzător factorului) sau mai mulți, separați prin operatorul \* (în acest caz, arborele corespunzător termenului conține în rădăcină operatorul \*, are ca subarbore stâng subarborele corespunzător primului factor, iar ca subarbore drept, arborele corespunzător succesiunii de factori care urmează). Observați că funcția Termen() este atât direct, cât și indirect recursivă.

În sfârșit, funcția CreareArbore(i) creează, într-un mod similar funcției Termen(), arborele corespunzător expresiei aritmetice care începe la poziția i.

```

struct Nod
{
    char inf;
    struct Nod *st, *dr; } ;
typedef struct Nod *Arbore;
Arbore rad;
char e[LgMax]; //expresia
int lg; //lungimea expresiei
Arbore CreareArbore(int & i);
Arbore Factor(int & i);
Arbore Termen(int & i);
Arbore Factor(int & i)
//construiește arborele binar asociat unui factor
Arbore p;
if (e[i]==‘(’)
    {++i;
     p=CreareArbore(i);
     i++;
     return p;}
p=new Nod; p->inf=e[i++]; p->st=p->dr=NULL;
return p;
}

Arbore Termen(int & i)
//construiește arborele binar asociat unui termen
Arbore p, r1;
r1=Factor(i);
if (i>lg || e[i]!='*') return r1;
p=new Nod;
p->inf=e[i++]; p->st=r1; p->dr=Termen(i);
return p;
}

Arbore CreareArbore(int & i)
//construiește arborele binar asociat expresiei aritmetice
Arbore p, r1;
if (i>lg) return NULL;
r1=Termen(i);
if (i>lg || e[i]==‘)’) return r1;
p=new Nod;
p->inf=e[i++]; p->st=r1; p->dr=CreareArbore(i);
return p;
}

```

### Exerciții propuse

1. Construji un arbore binar pentru fiecare din expresiile următoare și determinați notația poloneză.

$$\begin{array}{l} a^*b^*c+2^*d^*(a+1+2^*b^*c) \\ a+b+c+d+2 \end{array}$$

$$\begin{array}{l} 2^*a^*c^*(a^*b+c+3^*d+a^*b^*c^*d) \\ 2^*(a+b)^*(c+d+3)+5^*h^*k \end{array}$$

2. Scrieți o funcție care să evaluateze o expresie aritmetică. Funcția va primi ca parametru adresa rădăcinii arborelui asociat expresiei și numele fișierului în care sunt specificate valorile variabilelor. În fișier, pentru fiecare variabilă care apare în expresie, se află o linie pe care este scris numele variabilei și valoarea acesteia (un număr natural), separate printr-un singur spațiu.
3. Scrieți o funcție care să afișeze o expresie aritmetică în formă „uzuală”. Funcția va primi ca parametru adresa rădăcinii arborelui asociat expresiei. La afișarea expresiei se vor scrie doar parantezele rotunde strict necesare.

### 3.13. Aplicații rezolvate

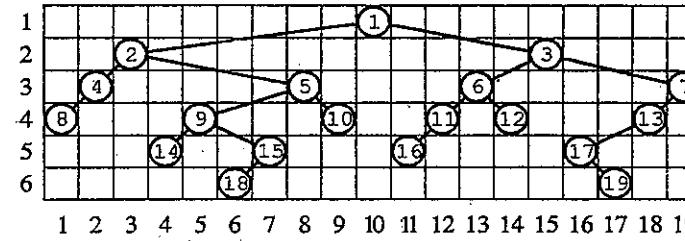
#### Lățime

Să presupunem că dorim să desenăm un arbore binar pe o foaie de matematică (în care am numerotat liniile de sus în jos și coloanele de la stânga la dreapta).

Desenul trebuie să respecte următoarele reguli:

- Nodurile situate pe același nivel trebuie să fie desenate pe aceeași linie. Nivelul rădăcinii arborelui este 1 și rădăcina va fi desenată pe linia 1. Fiii rădăcinii constituie nivelul 2 și vor fi desenați pe linia 2. Fiii filor rădăcinii constituie nivelul 3 și vor fi desenați pe linia 3 etc.
- Pe fiecare coloană poate fi desenat un singur nod.
- Nodurile din subarborele stâng al unui nod trebuie să fie desenate în coloanele din stânga coloanei în care este desenat nodul respectiv, iar nodurile din subarborele drept trebuie să fie desenate în coloanele din dreapta coloanei în care este desenat nodul respectiv.
- Orice coloană cuprinsă între coloana cea mai din stânga, în care este desenat un nod, și coloana cea mai din dreapta, în care este desenat un nod, trebuie să conțină un nod.

Să notăm cu L cea mai din stânga coloană în care este desenat un nod și cu R cea mai din dreapta coloană în care este desenat un nod. Lățimea arborelui binar este  $R-L+1$ . De exemplu:



Observați că lățimea acestui arbore binar este 19.

Putem calcula lățimea pe fiecare nivel ca fiind diferența dintre numărul celei mai din dreapta coloane care conține un nod de pe nivelul respectiv și numărul celei mai

din stânga coloane care conține un nod de pe nivelul respectiv +1. Arborele din exemplu are 6 niveluri, lățimile nivelurilor fiind 1, 13, 18, 18, 13, 12. Când desenam un arbore binar după aceste reguli ne interesează și lățimea nivelului cu lățime maximă și care este acest nivel. Dacă există mai multe niveluri cu lățimea maximă, ne interesează cel cu numărul cel mai mic.

În exemplul precedent, lățimea maximă a unui nivel este 18 și ea este atinsă pe nivelurile 3 și 4, deci nivelul care ne interesează este 3.

#### Cerință

Dat fiind un arbore binar, scrieți un program care să determine lățimea maximă a unui nivel în arborele desenat după regulile de mai sus și nivelul cu numărul cel mai mic care are această lățime.

#### Date de intrare

Fișierul de intrare latime.in conține pe prima linie un număr natural N reprezentând numărul de noduri din arbore. Nodurile sunt numerotate de la 1 la N. Fiecare dintre următoarele N linii conține câte 3 numere întregi separate prin căte un spațiu x s d cu semnificația „fiul stâng al nodului x este s, iar fiul drept al nodului x este d”. Dacă un nod nu are fiu stâng sau fiu drept, atunci s, respectiv d, pentru nodul respectiv vor avea valoarea -1. Rădăcina arborelui este nodul 1.

#### Date de ieșire

Fișierul de ieșire latime.out va conține o singură linie pe care se află două numere naturale separate printr-un singur spațiu, niv lat, cu semnificația „cel mai mic nivel de lățime maximă este niv, iar lățimea lui este lat”.

#### Restricții

$1 \leq N \leq 1000$

#### Exemplu

latime.in

```
19
1 2 3
2 4 5
3 6 7
4 8 -1
5 9 10
6 11 12
7 13 -1
8 -1 -1
9 14 15
10 -1 -1
11 16 -1
12 -1 -1
13 17 -1
14 -1 -1
15 18 -1
16 -1 -1
17 -1 19
18 -1 -1
19 -1 -1
```

latime.out

```
3 18
```

(.campion, 2004)

#### Soluție

Parcurg arborele pentru a afla pentru fiecare nivel coloana minimă și cea maximă. Apoi determin lățimea maximă și cel mai mic nivel cu această lățime.

```
#include <stdio.h>
#define infile "latime.in"
#define outfile "latime.out"

int arb[1000][2], col[1000], min[1000], max[1000];
/* arb[a][0]=fiul stang al nodului a
   arb[a][1]=fiul drept al nodului a
   minmax[i][0]=coloana cea mai din stanga de pe nivelul i
   minmax[i][1]=coloana cea mai din dreapta de pe nivelul i
   col[i]+1=coloana in care va fi desenat nodul i
   min[i]=coloana cea mai din stanga pe care se afla un nod
   de pe nivelul i
   max[i]=coloana cea mai din dreapta pe care se afla un nod
   de pe nivelul i
*/
int num; /*numarul de noduri din arbore */
int lat, niv;

void citire(void);
void rezolva(void);
void afisare(void);

int main()
{
    citire();
    rezolva();
    afisare();
    return 0;
}

void citire(void)
{
    FILE *fp=fopen(infile, "r");
    int a, i;
    fscanf (fp, "%d", &num);
    for (i=0; i<num; i++)
        { min[i]=0x7fffffff;
          max[i]=0; }
    for (i=0; i<num; i++)
        { fscanf (fp, "%d", &a);
          a--;
          fscanf (fp, "%d %d", &arb[a][0], &arb[a][1]);
          if (arb[a][0]>0) arb[a][0]--;
          if (arb[a][1]>0) arb[a][1]--;
        }
    fclose(fp);
}
```

```

int nr=0;
void parcurgere(int x, int nivx)
{ if (arb[x][0]!=-1) /* nodul x are fiu stang? */
    parcurgere(arb[x][0], nivx+1);
/*parcurg subarborele stang
compar cu coloana lui x cu coloana cea mai din stanga,
respectiv cea mai din dreapta de pe nivelul nivx */
if (min[nivx]>nr) min[nivx]=nr;
if (max[nivx]<nr) max[nivx]=nr;
/* coloana pe care se afla nodul x este 1+numarul de
noduri parcuse inaintea lui x */
col[x]=nr++;
if (arb[x][1]!=-1) /*nodul x are subarbore drept? */
    parcurgere(arb[x][1], nivx+1);
/* parcurg subarborele drept */
}

void rezolva()
{int i;
parcurgere(0, 0);
lat=-1;
for (i=0; i<num; i++)
    if (lat<max[i]-min[i])
        (lat=max[i]-min[i]; niv=i; )
}

void afisare()
{FILE *fp=fopen (outfile, "w");
fprintf (fp, "%d %d\n", niv+1, lat+1);
fclose(fp);
}

```

### *Interclasarea optimală a n siruri ordonate*

Fie  $n$  secvențe ordonate  $S_1, S_2, \dots, S_n$  de lungimi  $L_1, L_2, \dots, L_n$ . Să se interclasze cele  $n$  secvențe executând un număr minim de comparații între două elemente.

#### *Soluție*

Să notăm  $m=L_1+L_2+\dots+L_n$ . Vom construi un arbore de selecție.

Numim *arbore de selecție* un arbore binar complet în care fiecare nod este cel mai mic dintre fiili săi.

Fiecare nod din arbore are asociată ca valoare minimumul valorilor nodurilor din subarborele corespunzător.

Vom construi arboarele de selecție în mod *bottom-up*, apoi la fiecare pas selectăm minimumul care este rădăcina arborelui și restructurăm arborele. Construcția arborelui de selecție are complexitatea  $O(n)$ , restructurarea arborelui, care se repetă de  $m$  ori, are complexitatea  $O(\log n)$ . Deci algoritmul va avea complexitatea  $O(m \log n)$ .

Arborele de selecție este complet, deci vom utiliza reprezentarea secvențială.

```

#include <iostream.h>
#define MAXINT 32000
#define LMax 50
#define NMax 20

int n, m;
int L[NMax]; //lungimile secvențelor
int o[NMax*LMax]; //rezultatul interclasării
int s[NMax][LMax]; //secvențele
int A[2*NMax]; //arborele de selecție
int j[NMax]; //indicii curenti in secvențe

void Citire()
{ifstream fin("int.in");
int i, j;
fin>>n;
for (i=1; i<=n; i++)
    {fin>>L[i]; m+=L[i];
    s[i][L[i]+1]=MAXINT; }
for (i=1; i<=n; i++)
    for (j=1; j<=L[i]; j++)
        fin>>s[i][j];
}

void ConstrArbSel()
{int i;
for (i=n; i<2*n; i++) A[i]=s[i-n+1][1];
//initializarea nodurilor terminale
for (i=n-1; i>0; i--)
    if (A[2*i]<A[2*i+1]) A[i]=A[2*i];
    else A[i]=A[2*i+1];
//initializez valorile indicilor in secvențe
for (i=1; i<=n; i++) j[i]=1;
}

void Restructurare()
{int i=1, tata, frate;
//determin secvența corespunzătoare nodului eliminat
while (i<=n-1)
    if (A[i]==A[2*i]) i=2*i;
    else i=2*i+1;
j[i-n+1]++; A[i]=s[i-n+1][j[i-n+1]];
while (i>1)
    {tata=i/2;
    if (i==2*tata) frate=2*tata+1;
    else frate=2*tata;
    if (A[i]>A[frate]) A[tata]=A[frate];
    else A[tata]=A[i];
    i=tata;
    }
}

```

```

int main()
{
    int k;
    Citire();
    ConstrArbSel();
    for (k=1; k<=m; k++)
        { o[k]=A[1];
          Restructurare(); }
    for (k=i; k<=m; k++)
        cout<<o[k]<< ' ';
    cout<<endl;
    return 0;
}
  
```

### Piloți

Vasile are o companie de transport aerian. Pentru a se menține pe piață, el trebuie să reducă cheltuielile cât mai mult posibil.

La compania sa există N piloți (N par). Pilofii sunt numerotați de la 1 la N în ordinea crescătoare a vîrstei (pilotul 1 este cel mai tânăr, pilotul N cel mai bătrân).

Vasile trebuie să constituie  $N/2$  echipaže. Un echipaj este format din 2 piloți (căpitanul și asistentul său). Căpitanul trebuie să fie mai în vîrstă decât asistentul său. În contractul fiecărui pilot sunt prevăzute două salarii: unul pentru cazul în care el este căpitan al echipajului, celălalt pentru cazul în care el este asistent. Evident, pentru orice pilot salariau său de căpitan este mai mare decât salariau său de asistent.

Salariile pot să difere de la un pilot la altul. Chiar se poate întâmpla că salariau căpitanului să fie mai mic decât salariau asistentului său.

Pentru a cheltui cât mai puțini bani pe salariile piloților, Vasile trebuie să determine o distribuire optimală a piloților pe echipaje.

### Cerință

Scrieți un program care să determine suma minimă necesară pentru a plăti salariile piloților.

### Date de intrare

Fișierul de intrare piloti.in conține pe prima linie un număr natural N reprezentând numărul de piloți. Pe următoarele N linii sunt informații despre salariile piloților. Pe linia  $i+1$  se află două numere naturale c separate printr-un spațiu (c reprezintă salariau pilotului  $i$  pe post de căpitan, iar c reprezintă salariau pilotului  $i$  pe post de asistent).

### Date de ieșire

Fișierul de ieșire piloti.out va conține o singură linie pe care va fi afișată suma minimă necesară pentru a plăti salariile celor N piloți.

### Restricții

$2 \leq N \leq 10000$ ; N par

Pentru orice pilot  $1 \leq a < c \leq 100000$ .

### Exemplu

piloti.in	piloti.out	piloti.in	piloti.out
6	32000	6	33000
10000 7000		5000 3000	
9000 3000		4000 1000	
6000 4000		9000 7000	
5000 1000		11000 5000	
9000 3000		7000 3000	
8000 6000		8000 6000	

(campion, 2005)

### Soluție

Vom utiliza metoda de programare Greedy. Primul pilot trebuie să fie asistent. Dintre piloții 2 și 3, unul trebuie să fie asistent. Dintre piloții 4, 5 și cel care a rămas dintre 2 și 3, unul trebuie să fie asistent. Deci la fiecare doi piloți citiți, trebuie să aleg unul care să fie asistent (alegerea se face dintre toți cei citiți în prealabil și care nu au fost deja stabiliți ca asistenți).

Criteriul de alegere: diferența dintre salariau său de căpitan și salariau său de asistent să fie maximă.

Vom utiliza variabila rez, în care reținem suma totală cheltuită pentru salariile piloților. Când citim un pilot, voi considera că el este căpitan și voi adăuga salariau lui de căpitan la rez, iar diferența dintre salariau său de căpitan și salariau său de asistent o voi insera într-un *max-heap* H. Dar după fiecare două citiri trebuie să aleg un asistent. Voi extrage maximumul din *max-heap*-ul H. Din rez voi scădea maximumul (ceea ce înseamnă că pilotul corespunzător devine asistent).

```

#include <iostream.h>
#define InFile "piloti.in"
#define OutFile "piloti.out"
#define NMax 10001

int H[NMax], lg=0;

int maxheap_top()
{
    return H[1];
}

void maxheap_push(int x)
{
    int tata, fiu;
    lg++;
    fiu=lg; tata=lg/2;
    while (tata>0)
        if (x>H[tata])
            {H[fiu]=H[tata];
             fiu=tata;
             tata=tata/2; }
        else break;
    H[fiu]=x;
}
  
```

```

void maxheap_pop()
{int tata, fiu, x;
x=H[lg--];
tata=1; fiu=2;
while (fiu<=lg)
    {if (fiu<lg && H[fiu]<H[fiu+1]) fiu++;
     if (x>H[fiu])
        {H[tata]=H[fiu]; tata=fiu; fiu=2*fiu;}
     else break;
    }
H[tata]=x;
}

int main(void)
{int n, rez=0, capitan, asistent;
ofstream fout(OutFile);
ifstream fin(InFile);
for (fin>>n; n, --n)
    {fin>>capitan>>asistent;
     rez+=capitan;
     maxheap_push (capitan-asistent);
     if (n%2==0)
        {rez-=maxheap_top(); maxheap_pop();}
    }
fout<<rez<<endl;
fout.close();
return 0;
}

```

### 3.14. Aplicații propuse

#### 1. Kruskal

Implementați eficient algoritmul lui Kruskal de determinare a unui arbore parțial de cost minim al unui graf neorientat conex, organizând muchiile ca *min-heap* și utilizând operații Union-Find.

#### 2. Arbore

Să considerăm un arbore cu  $N$  vârfuri, numerotate de la 1 la  $N$ . Scrieți un program care să adauge, dacă este posibil, un număr minim de muchii astfel încât fiecare vârf să aparțină exact unui singur ciclu.

#### Date de intrare

Fișierul de intrare *arbore.in* conține pe prima linie numărul natural  $N$ . Pe următoarele  $N-1$  linii, sunt descrise muchiile arborelui, prin cele două extremități.

#### Date de ieșire

Fișierul de ieșire *arbore.out* va conține pe prima linie valoarea  $-1$  dacă problema nu admite soluție, respectiv numărul de muchii adăugate, dacă problema

admete soluție. Dacă problema admite soluție, pe fiecare dintre următoarele linii se vor scrie extremitățile unei muchii adăugate, separate printr-un spațiu.

#### Restricții

$3 \leq N \leq 100$

#### Exemple

<i>arbore.in</i>	<i>arbore.out</i>	<i>arbore.in</i>	<i>arbore.out</i>
4	-1	7	2
1 2		1 2	6 7
2 3		1 3	4 2
2 4		3 5	
		3 4	
		5 6	
		5 7	

(Olimpiada Națională de Informatică, Brăila, 2002)

#### 3. Barca

Fiind o frumoasă zi de primăvară,  $n$  pitici au luat o barcă și au plecat la plimbare pe lac. În timpul plimbării, ei întâlnesc alți  $m$  pitici care se scaldă; fiecare pitic întâlnit e urcat în barcă. Dacă barca începe să se scufunde din cauza greutății prea mari, este aruncat în apă cel mai greu pitic (un pitic aruncat nu mai e luat în barcă). Dacă sunt mai mulți pitici cu aceeași greutate, este aruncat cel care a stat cel mai mult în barcă. În fișierul de intrare, piticii sunt scriși în ordinea în care urcă în barcă.

#### Cerință

Aflați al cătelea a fost aruncat în apă fiecare pitic dintr-un număr de  $10$  pitici precizați. Se cunoaște greutatea fiecărui pitic și greutatea maximă suportată de barcă. Piticii sunt identificați prin numere naturale de la  $1$  la  $n+m$ .

#### Date de intrare

Pe prima linie a fișierului de intrare *barca.in* sunt scrise  $3$  numere naturale:  $n$  – numărul piticilor care pleacă la plimbare,  $g$  – greutatea maximă suportată de barcă,  $m$  – numărul piticilor întâlniți pe lac. Pe următoarea linie sunt  $10$  numere naturale cuprinse între  $1$  și  $n+m$ , reprezentând indicii piticilor pentru care trebuie aflată ordinea în care au fost aruncați. Pe următoarele  $n$  linii sunt scrise greutățile piticilor aflați în barcă la început, iar pe ultimele  $m$  linii, greutățile piticilor aflați în apă, în ordinea în care sunt întâlniți. Numerele de pe aceeași linie sunt separate de către un spațiu.

#### Date de ieșire

Fișierul de ieșire *barca.out* va conține  $10$  linii; pe fiecare linie se află căte un număr care precizează al cătelea a fost aruncat din barcă fiecare dintre cei  $10$  pitici, respectând ordinea din fișierul de intrare. Pentru un pitic care nu e aruncat se va scrie numărul  $0$ .

#### Restricții

$1 \leq n, m \leq 100000$

$1 \leq g \leq 1000000000$

Greutatea unui pitic este un număr natural din intervalul  $[1, 10000]$ .

**Exemplu**

```
barca.in
4 20 12
1 2 3 4 5 6 7 8 10 11
7
8
5
2
6
3
3
9
5
6
4
11
6
6
2
5
```

```
barca.out
2
1
6
0
4
0
0
3
5
0
```

(campion, 2005)

**4. Formă normală**

Fie  $E$  o expresie care respectă următoarele condiții :

- poate conține numai operatorii  $+$  (adunare) și  $*$  (înmulțire);
- operanții pot fi doar nume de variabile formate dintr-o singură literă mică a alfabetului englez;
- poate conține paranteze rotunde.

Stiind că puteți aplica următoarele proprietăți :

1. Comutativitatea adunării :  $a+b=b+a$ ,  $\forall a, b$ ;
- 1'. Comutativitatea înmulțirii :  $a*b=b*a$ ,  $\forall a, b$ ;
2. Asociativitatea adunării :  $a+(b+c)=(a+b)+c=a+b+c$ ,  $\forall a, b, c$ ;
- 2'. Asociativitatea înmulțirii :  $a*(b*c)=(a*b)*c=a*b*c$ ,  $\forall a, b, c$ ;
3. Distributivitatea înmulțirii față de adunare :  $a*(b+c)=a*b+a*c$ ,  $\forall a, b, c$ ,

aduceți expresia dată în formă normală.

Forma normală a unei expresii este o sumă de produse, echivalentă cu expresia dată, astfel încât atât termenii sumei, cât și factorii din fiecare termen al sumei sunt ordonați alfabetic.

**Date de intrare**

Fișierul de intrare `expresie.in` conține pe prima linie o expresie de cel mult 1000 de caractere, corectă în raport cu restricțiile enunțate mai sus. Expressia nu conține spații.

**Date de ieșire**

Pe prima linie a fișierului `expresie.out` va fi afișată expresia în formă normală.

**Exemplu**

```
expresie.in
x*a+c*(a+b*c*d)
```

```
expresie.out
a*c+a*x+b*c*c*d
```

(Olimpiada Națională de Informatică, Mediaș, 1999)

**5. asmin**

Se consideră un arbore (graf conex aciclic) cu  $N$  vârfuri, fără rădăcină fixată. Drept rădăcină poate fi ales oricare dintre vârfuri. Să presupunem că a fost ales vârful cu numărul  $T$ . Între oricare vârf și  $T$  există un drum unic care conține fiecare vârf al arborelui cel mult o singură dată (un drum între vâfurile  $i$  și  $j$  este o secvență de vârfuri, care începe cu  $i$ , se termină cu  $j$ , iar între oricare două vârfuri consecutive există o muchie în arbore). Fiecare vârf  $i$  (inclusiv  $T$ ) trebuie să i se asocieze o valoare  $V_i$ , mai mare sau egală cu 0, astfel încât suma valorilor vâfurilor de pe drumul dintre  $i$  și rădăcina  $T$ , împărțită la  $k$ , să dea restul  $R_i$ . Se definește costul arborelui cu rădăcina fixată în  $T$ ,  $C_T$  ca fiind suma valorilor asociate fiecarui nod. Dintre toate posibilitățile de alegere a valorilor  $V_i$  care respectă condiția precizată anterior, se va alege aceea pentru care  $C_T$  este minim.

Se constată ușor că alegând alt vârf drept rădăcină, de exemplu, vârful  $S$  (diferit de  $T$ ),  $C_S$  nu este neapărat egal cu  $C_T$ .

**Cerință**

Dându-se un arbore cu  $N$  vârfuri, un număr întreg  $K$  și valorile  $R_i$ ,  $i=1, 2, \dots, N$ , corespunzătoare fiecărui vârf, determinați acele vârfuri  $T$ , care pot fi alese drept rădăcină, pentru care costul  $C_T$  este minim (adică  $C_T \leq C_S$ , oricare ar fi  $S$  diferit de  $T$ ), precum și costul respectiv.

**Date de intrare**

Pe prima linie a fișierului de intrare `asmin.in` se află două valori întregi :  $N$  și  $K$ . Pe următoarele  $N-1$  linii se află câte două numere întregi  $a$  și  $b$ , separate printr-un spațiu, având semnificația că există muchie între vâfurile  $a$  și  $b$ . Vâfurile sunt numerotate de la 1 la  $N$ . Pe următoarea linie se află  $N$  numere întregi, reprezentând valorile  $R_i$ ,  $i=1, 2, \dots, N$ .

**Date de ieșire**

Pe prima linie a fișierului de ieșire `asmin.out` se vor afișa două valori întregi :  $c$  și  $M$ .  $C$  reprezintă costul minim posibil al arborelui.  $M$  reprezintă numărul de vârfuri care pot fi alese drept rădăcină și pentru care se obține costul  $c$ . Pe a doua linie se află  $M$  numere întregi separate prin câte un spațiu, scrise în ordine crescătoare, reprezentând numerele vâfurilor ce pot fi alese ca rădăcină, astfel încât să se obțină costul  $c$ .

**Restricții și precizări**

$2 \leq N \leq 16000$

$2 \leq K \leq 1000$

$0 \leq R_i \leq K-1$

**Exemplu**

```
asmin.in
5 3
1 2
1 3
2 4
2 5
0 1 2 1 0
```

```
asmin.out
5 2
1 5
```

(Olimpiada Națională de Informatică, 2003)



**Date de ieșire**

Fișierul de ieșire `aclor.out` va conține o singură linie pe care va fi scris numărul de picturi frumoase care se pot obține pentru arborele dat.

**Restricții și precizări**

$0 < N \leq 100\,000$ ,  $1 \leq R \leq N$ ,  $1 \leq K \leq 100$

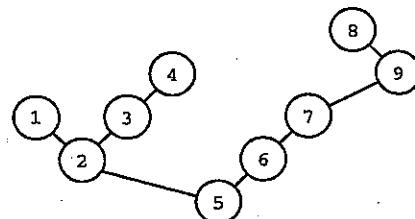
**Exemplu**  
`aclor.in`

```
9 5 4
0 0
1 3
0 4
0 0
2 6
0 7
0 9
0 0
8 0
```

`aclor.out`

3601

*Figură*



(Olimpiada Națională de Informatică, Tîrgoviște, 2006)

## 4. Pattern-matching

### 4.1. Introducere

Fie  $T$  un sir format din  $n$  caractere (denumit text) și  $P$  un alt sir format din  $m$  caractere (denumit model sau pattern). Spunem că  $P$  apare în  $T$  cu deplasamentul  $s$  ( $0 \leq s \leq n-m$ ) dacă  $T[s+j] = P[j]$ ,  $\forall j$ ,  $0 \leq j < m$ .

Prin *pattern-matching* se înțelege determinarea tuturor deplasamentelor cu care modelul  $P$  apare în textul  $T$ .

**Exemplu**

Fie  $T = "abcabaabcbabac"$  și  $P = "abaa"$ . Pentru  $s=3$  se obține:

```
abcabaabcbabac
  s abaa
```

### 4.2. Algoritmul elementar (naiv)

Algoritmul elementar (naiv) presupune parcurgerea tuturor deplasamentelor posibile și verificarea potrivirii modelului în text.

Algoritmul în pseudocod este următorul:

```
n ← lungime(T)
m ← lungime(P)
pentru s ← 0, n-m execută
  dacă P[0..m-1] = T[s..s+m-1] atunci
    scrie "am gasit subsirul in pozitia ", s
```

Timpul de execuție este  $O((n-m+1)*m)$ , deoarece există  $n-m+1$  valori posibile pentru  $s$ , iar pentru fiecare  $s$ , testarea  $P[0..m-1] = T[s..s+m-1]$  implică un ciclu cu  $m$  pași. Algoritmul este ineficient, deoarece nu folosește absolut deloc informațiile obținute prin prelucrarea pentru o valoare oarecare a lui  $s$  în prelucrările ulterioare. Astfel, dacă pentru o anumită valoare a lui  $s$  în timpul verificării celor  $m$  caractere apare unul care „nu se potrivește”, verificarea poate fi abandonată și se poate trece la următoarea valoare a lui  $s$ . În același timp, dacă, pentru un anumit deplasament  $s$  analiza lui  $P$  ne indică faptul că anumite deplasamente nu pot furniza un rezultat pozitiv, se poate renunța la ele. De exemplu, fie  $T = "aaabaab..."$  și  $P = "aaab"$ . Se observă că pentru  $s=0$  se obține o căutare cu rezultat pozitiv, dar, în același timp, din studierea lui  $P$

se observă că nici unul dintre deplasamentele  $s=1, 2, 3$  nu va furniza un rezultat pozitiv deoarece  $P[3]='b'$ , deci nu are rost să fie testate aceste valori pentru deplasament.

### 4.3. Algoritmul Knuth-Morris-Pratt (KMP)

Ideea algoritmului KMP este de a defini o funcție (numită *funcție prefix*) care să rețină informații despre *potrivirea modelului cu deplasamentele lui*.

Funcția este definită printr-un tablou cu  $m$  elemente, tablou care reține în elementul de indice  $x$  o valoare care indică ce deplasament se va testa în continuare, știind că primele  $x$  elemente din model „se potrivesc” cu  $x$  elemente din text pentru deplasamentul curent.

Formalizând afirmația de mai sus, problema care se pune este următoarea:

Știind că  $T[s..s+x-1]=P[0..x-1]$ , deci  $x$  elemente se „potrivesc”, care este cel mai mic deplasament  $s'>s$ , astfel încât  $T[s'..s'+k-1]=P[0..k-1]$ , cu  $s+x=s'+k$ . În cel mai bun caz vom avea  $s'=s+x$ , ceea ce indică faptul că deplasamentele  $s+1, s+2, \dots, s+x-1$  nu trebuie testate.

Tabloul care definește funcția se obține prin compararea modelului  $P$  cu el însuși, în modul următor: deoarece  $T[s'..s'+k-1]$  face parte din porțiunea de text cunoscută, el este sufix al sirului  $P[0..x-1]$ .

#### Exemplu

Să considerăm  $T="bacbababaabcbab"$  și  $P="ababaca"$ .

T	bacbababaabcbab	$s=4$
P	<u>s=4</u> ababaca	$x=5$
		
T	bacbababaabcbab	$s'=6$
P	<u>s'=6</u> ababaca	$x=3$
		
$P[0..4]$	ababa	
$P[0..2]$	aba	
		$P[0..2]$ este sufix al lui $P[0..4]$

adică cel mai lung prefix al lui  $P$  care este și sufix al lui  $P[0..4]$  este  $P[0..2]$ ; deci în tabloul care reprezintă funcția vom avea  $Urm[4]=2$ , cu interpretarea: *dacă la deplasamentul  $s$  s-au potrivit cu succes 5 caractere, următorul deplasament posibil corect este  $s'=s+(4-Urm[4])=s+(4-2)=s+2$ .*

Prin urmare, funcția prefix  $Urm$  se definește astfel:

$Urm: \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$

$Urm[x]=\max\{k, k < x \text{ și } P[0..k-1]$ , care este prefix al lui  $P$ , este și sufix al lui  $P[0..x-1]\}=\text{lunimea celui mai lung prefix al lui } P \text{ care este sufix al lui } P[0..x-1]$ .

#### Exemplu

Considerăm modelul  $P="ababababca"$ . Construind pas cu pas vectorul  $Urm$  vom obține:

x	P	$P[0..x]$	k
0	ababababca	a	0
1	ababababca	ab	0
2	ababababca	aba	1
3	ababababca	abab	2
4	ababababca	ababa	3
5	ababababca	ababab	4
6	ababababca	abababa	5
7	ababababca	abababab	6
8	ababababca	ababababc	1
9	ababababca	ababababcă	1

Timpul de execuție al acestui algoritm este  $O(n+m)$ , deoarece calculul tabloului  $Urm$  (funcția prefix) consumă  $O(m)$ , iar determinarea „potrivirilor” consumă  $O(n)$ .

```
#include <iostream.h>
#include <string.h>
#define nMax 1000
#define mMax 255

int Urm[mMax];
unsigned char T[nMax], P[mMax];
int n, m;

void Urmatorul(char *P)
{
    int k=-1, x;
    Urm[0]=0;
    for (x=1; x<m; x++)
        {while (k>-1 && P[k+1]!=P[x]) k=Urm[k];
         if (P[k+1]==P[x]) k++;
         Urm[x]=k; }
}

int main()
{
    int i, x=-1;
    ifstream f("KMP.IN");
    ofstream g("KMP.OUT");
    f.getline(T, 1000); f.getline(P, 255);
    n=strlen(T); m=strlen(P);
    Urmatorul(P);
    for (i=0; i<n; i++)
        {while (x>-1 && P[x+1]!=T[i]) x=Urm[x];
         if (P[x+1]==T[i]) x++; //s-a potrivit
         if (x==m-1)
             {g<<"Am gasit subsirul in pozitia "<<i-m+1<<endl;
              x=Urm[x]; }
      }
    g.close();
    return 0;
}
```

## 4.4. Aplicație. Deserț

Fie o imagine alb-negru care reprezintă fotografia, realizată din satelit, a unei zone dintr-un deserț unde se caută o construcție secretă de formă dreptunghiulară. Imaginea porțiunii de deserț analizate a fost codificată într-o matrice având dimensiunile maxime de  $N \times 255$  elemente. Imaginea construcției este codificată într-o matrice având  $K \times 32$  elemente. Elementele celor două matrice pot fi numai caracterele '.' și '#'.

Determinați de câte ori se regăsește fotografia construcției pe hartă.

### Date de intrare

Fișierul de intrare `desert.in` conține pe prima linie două numere naturale nenule  $N$  și  $K$ ,  $N$  reprezentând numărul de linii ale matricei care codifică fotografia deserțului, iar  $K$  numărul de linii ale matricei care codifică fotografia construcției. Următoarele  $K$  linii conțin, în formă codificată, matricea fotografiei construcției (caractere '#' și '.' neseparate prin spații).

Următoarele  $N$  linii conțin, în formă codificată, matricea fotografiei deserțului (caracter ' #' și '.' neseparate prin spații).

### Date de ieșire

Fișierul de ieșire `desert.out` va conține o singură linie pe care va fi scris un număr natural reprezentând numărul de potriviri ale matricei fotografiei construcției pe hartă.

### Restricții și precizări

$$3 \leq N \leq 1024$$

$$1 < K < N$$

### Exemplu

`desert.in`

3 2

#.....(în total 31 de puncte)...  
#.....(în total 31 de puncte)...  
#.....(în total 254 de puncte)...  
#.....(în total 254 de puncte)...  
#.....(în total 254 de puncte)...

`desert.out`

2

(Baraj selecție lot național, Bacău, 2001)

### Soluție

Având în vedere faptul că în matricea construcției, o linie are exact 32 caractere, și acestea sunt doar de două tipuri '.', sau '#', ele pot fi considerate valori 0 și 1, iar cele 32 de cifre binare rezultante, ca fiind reprezentarea în baza 2 a unui număr întreg de tip `long`. Deci cele  $K$  linii ale construcției pot fi considerate ca un model format din  $K$  numere întregi de tip `long`.

Pentru determinarea aparițiilor matricei construcției în matricea `desert` se utilizează algoritmul KMP. Este folosit, în plus, un vector `poz`, ce reține pozițiile orizontale în care apar posibilele potriviri.

```
#include <stdio.h>
#include <string.h>
char s[260];
int q, x, y, z, n, nc;
long v[1030], l1;
int Urm[1030], poz[230];
unsigned long contor;
FILE *f, *g;
long s21(char *s) //codifica constructia pe biti
{ long l=0;
  int i;
  for (i=0; i<32; i++)
    { l<<=1;
      if (s[i]=='#') l|=1;    }
  return l;
}

int main()
{ char c;
  f=fopen("desert.in", "r");
  fscanf(f, "%d %d%c", &n, &nc, &c);
  for (q=1; q<=nc; q++)
    { fgets(s, 260, f);
      v[q]=s21(s);          }
  for (x=2; x<=nc; x++) //se construiește vectorul Urm
  { y=Urm[x-1];
    if (v[x]==v[y+1])
      Urm[x]=Urm[x-1]+1;
    else
      if (v[x]==v[1]) Urm[x]=1;
  }
  for (q=1; q<=n; q++) //analizez cele n linii ale fotografiei
  { fgets(s, 260, f); //citesc o linie
    l1=s21(s); //codific primele 32 caractere
    for (x=0; x<=223; x++) //parcurg cele 255-32 caractere
    { y=poz[x];
      if (v[y+1]==l1)
        { y++; //am mai gasit o potrivire - o numar
          if (y==nc) //am detectat o constructie
            { contor++; //o numar
              y=Urm[y];}
        }
      else //daca NU caut alt deplasament
        while ((y>0) && (v[y+1]!=l1)) y=Urm[y];
      poz[x]=y;
      l1<<=1;
      l1|=x<223 && s[x+32]=='#'; //codific ultimul bit
    }
  }
  fclose(f);
  g=fopen("desert.out", "w");
  fprintf(g, "%lu\n", contor);
  fclose(g); return 0;
}
```

## 4.5. Probleme propuse

### 1. Colier

Maria a primit de ziua ei de la prietene mai multe cadouri. Printre acestea ea a găsit și un colier format din mărgele colorate în mai multe culori. Privind colierul, Maria și-a pus problema dacă nu cumva o anumită secvență de culori apare distinct de mai multe ori în colier și unde anume.

#### Date de intrare

Prima linie a fișierului de intrare `colier.in` conține culorile colierului, reprezentate ca un sir de caractere de lungime  $N$ , caracterele fiind litere mici ale alfabetului. Cea de-a doua linie conține sirul de caractere litere mici care reprezintă secvența de culori căutată.

#### Date de ieșire

Prima linie a fișierului de ieșire `colier.out` va conține o valoare naturală  $K$ , reprezentând numărul de apariții ale secvenței de culori sau valoarea -1, în cazul în care secvența de culori nu apare în colier. În cazul în care secvența de culori apare cel puțin o dată, linia a doua a fișierului de ieșire va conține  $K$  valori naturale separate prin câte un spațiu, care reprezintă pozițiile în care apare secvența de culori, considerând că pozițiile mărgezelor în colier sunt numerotate începând cu 1.

**Restricții**  
 $1 \leq M \leq N \leq 32000$

#### Exemplu

colier.in	colier.out
baababacabcabaccababacaccccaaba	4 3 12 17 28

### 2. Cifru

Copiii solarieni se joacă adesea trimîndu-și mesaje codificate. Pentru codificare, ei folosesc un cifru bazat pe o permutare  $p$  a literelor alfabetului solarian și un număr natural  $d$ . Alfabetul solarian conține  $m$  litere foarte complicate, așa 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 și  $m$ ,  $c_1 c_2 \dots c_n$ , codificarea mesajului se realizează astfel: se înlocuiește fiecare literă  $c_i$  cu  $p(c_i)$ , apoi sirul obținut  $p(c_1) p(c_2) \dots p(c_n)$  se rotește spre dreapta, făcând o permutare circulară cu  $d$  poziții, rezultând sirul  $p(c_{n-d+1}) \dots p(c_{n-1}) p(c_n) p(c_1) p(c_2) \dots p(c_{n-d})$ .

De exemplu, fie mesajul 2 1 3 3 2 1, permutarea  $p=(3 \ 1 \ 2)$  și  $d=2$ . Aplicând permutarea  $p$  vom obține sirul 1 3 2 2 1 3, apoi rotind spre dreapta sirul cu două poziții obținem codificarea 1 3 1 3 2 2.

#### Cerință

Date fiind un mesaj necodificat și codificarea sa, determinați cifrul folosit (permutarea  $p$  și numărul  $d$ ).

#### Date de intrare

Fișierul de intrare `cifru.in` conține pe prima linie numele naturale  $m$  și  $m$ , separate prin spațiu, reprezentând lungimea mesajului, 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 și  $m$  separate prin câte un spațiu. Pe cea de-a treia linie este scris mesajul codificat ca o succesiune de  $n$  numere cuprinse între 1 și  $m$  separate prin câte un spațiu.

#### Date de ieșire

Fișierul de ieșire `cifru.out` va conține pe prima linie numărul natural  $d$ , reprezentând numărul de poziții cu care s-a realizat permutarea circulară spre dreapta. Dacă pentru  $d$  există mai multe posibilități, 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 spațiu.

#### Restriții

$n \leq 100000$   
 $m \leq 9999$

Mesajul conține fiecare număr natural din intervalul  $[1, m]$  cel puțin o dată.

#### Exemplu

cifru.in	cifru.out
6 3	2
2 1 3 3 2 1	3 1 2
1 3 1 3 2 2	

(Olimpiada Națională de Informatică, Tîrgoviște, 2006)

sevențele binare de lungime p apar cu aceeași probabilitate ca ultimi p biți ai cheii, este de preferat să utilizăm o funcție în care se utilizează toți biții cheii.

Un număr prim apropiat de o putere a lui 2 este adesea o bună alegere pentru m.

### Observație

Potem considera orice funcție hash de forma  $h(x) = (ax+b) \bmod m$ , cu  $a \neq 0$ .

Exemplu de funcție hash pentru chei și sir de caractere :

```
int hashfunction(char *s)
{ int i;
  for (i=0; *s; s++) i = 131*i + *s;
  return( i % m ); }
```

### Metoda 2. Înmulțirea

O două metodă utilizează calcule aritmetice mai complexe: se înmulțește cheia x cu un număr subunitar  $0 < y < 1$ , se consideră apoi partea fracționară a produsului  $x \cdot y$ , care se înmulțește cu m, și se reține partea întreagă a rezultatului :

$$h(x) = [m \cdot (\text{frac}(x \cdot y))]$$

Un avantaj al acestei metode este că nu există valori „rele” pentru m, prin urmare, m poate fi o putere a lui 2 pentru a implementa eficient funcția hash.

### Metoda 3. Hashing universal

Dacă un adversar „răutăcios” va alege cheile astfel încât pentru o anumită funcție hash să aibă toate aceeași valoare hash, atunci... obținem un timp de execuție liniar.

Orice funcție hash fixată este vulnerabilă la o astfel de abordare. Soluția ar fi să alegem o funcție hash independentă de cheile ce urmează a fi stocate.

Ideea de bază este să selectăm funcția hash la întâmplare dintr-o listă predefinită de funcții hash la începutul execuției programului. Se numește familie universală de funcții hash o mulțime H de funcții definite pe A cu valori în  $\{0, 1, \dots, m-1\}$ , astfel încât pentru orice pereche de chei x, y ( $x \neq y$ ), numărul funcțiilor hash din familie pentru care  $h(x) = h(y)$  este cel mult  $|H|/m$ . Cu alte cuvinte, dacă alegem aleator o funcție din această familie, probabilitatea de a obține o coliziune (când funcția hash produce o aceeași valoare pentru două chei distincte) pentru cheile x și y este  $< 1/m$ .

### Exemplu practic

Alegem  $m=2^p$  și generăm aleator un număr natural impar r. Codul hash pentru un întreg x se află înmulțind pe x cu r și păstrând cei mai semnificativi p biți ai rezultatului (înmulțirea se face pe 32 de biți, ignorând depășirile). Această variantă aproximează bine o familie universală de funcții hash :

```
int r=(rand()<<1) | 1;
...
inline int hash(int x)
{ return((unsigned) (x*r)>>(32-p)); }
```

## 5. Hashing

### 5.1. Introducere

Multe aplicații necesită o structură dinamică de date, care să permită executarea eficientă a operațiilor *Insert*, *Delete* și *Search*. O astfel de structură de date poartă numele de *dicționar*.

Tabelele hash (tabele de dispersie) reprezintă o variantă de implementare a unui dicționar, foarte ușor de implementat și frecvent utilizată în practică. În cazul cel mai defavorabil, o tabelă hash se poate comporta ca o listă simplu înlănțuită (deci operațiile specificate se execută în timp liniar). Dar în practică, utilizând tabelele hash, cele 3 operații se execută eficient.

Să considerăm A o mulțime cu n valori (denumite chei), care trebuie stocate în structura de date, și T un vector cu m componente (denumit tabelă hash). Vom considera o funcție hash prin care se asociază fiecărei chei un număr întreg din intervalul  $[0, m-1]$ ,  $h: A \rightarrow \{0, 1, \dots, m-1\}$ .

Funcția hash are proprietatea că valorile ei sunt uniform distribuite în intervalul specificat. Aceste valori vor fi utilizate ca indici în vectorul T. Mai exact, cheia x va fi plasată în tabela hash pe poziția  $h(x)$ .

### 5.2. Funcții hash

O funcție hash bună trebuie să distribuie uniform cheile în tabela hash. Din păcate, această condiție e greu de îndeplinit, fiindcă nu cunoaștem structura cheilor. În general, funcțiile hash sunt definite pe mulțimea numerelor naturale (cheile sunt considerate numere naturale). În cazul în care cheile nu sunt numere naturale, ele pot fi transformate în numere naturale.

De exemplu, pentru o cheie sir de caractere, vom considera că sirul este un număr natural scris în baza 128 (pentru codul ASCII) sau 256 (pentru ASCII extins).

#### Metoda 1. Restul împărțirii la m

O primă metodă de a obține o valoare întreagă din intervalul  $[0, m-1]$  este:  $h(x) = x \% m$ . În acest caz este indicat să evităm ca m să fie de forma  $2^p$  (pentru că atunci  $h(x)$  ar avea ca valoare ultimii p biți ai lui x). Exceptând cazul în care știm că toate

### 5.3. Rezolvarea coliziunilor

Când funcția *hash* produce o aceeași valoare pentru două chei distincte se produce o *colizie* (astfel de chei sunt denumite sinonime). Prin urmare, este necesar și un mecanism de rezolvare a coliziunilor.

Există două metode principale de rezolvare a coliziunilor.

#### *Chaining (înlănțuire)*

Toate valorile sinonime (care au aceeași valoare *hash*) vor fi plasate într-o listă înlănțuită. Astfel, în vectorul *T* pe poziția *i* vom reține un pointer către începutul listei înlănțuite formate din toate valorile *x* pentru care *h(x)=i*.

Operația *Insert* se execută în  $O(1)$  (se inserează valoarea *x* la începutul listei *T[h(x)]*). Operația *Search* presupune căutarea elementului cu cheia *x* în lista înlănțuită *T[h(x)]*. Operația *Delete* presupune căutarea elementului cu cheia *x* în lista înlănțuită *T[h(x)]*, apoi ștergerea din listă. Ultimele două operații au timpul de execuție proporțional cu lungimea listei *T[h(x)]*. Analizând complexitatea în cazul cel mai defavorabil, lungimea listei poate fi  $O(n)$  (toate elementele sunt sinonime). Analizând complexitatea în medie, observăm că lungimea listei depinde de cât de uniform distribuie funcția *hash* cheile.

Dacă presupunem că funcția *hash* va distribui uniform valorile, atunci lungimea medie a listei este  $n/m$  (această valoare este denumită factor de încărcare a tabelei), deci timpul de execuție va fi  $O(1+n/m)$ .

#### *Open-addressing (adresare deschisă)*

În tabela *hash* nu sunt memorări pointeri, ci sunt memorate elementele mulțimii *A*. Astfel în tabelă se vor afla poziții ocupate cu elemente din *A* și poziții libere (acestea vor fi marcate cu o valoare specială pe care o vom denumi *NIL*).

În acest caz, pentru a insera un element *x* în tabela *hash* se vor executa mai multe încercări până când determinăm o poziție liberă, în care putem plasa elementul *x*.

Pozиțiile care se examinează depend de cheia *x*. În acest caz, funcția *hash* va avea doi parametri: cheia *x* și numărul încercării curente (numerotarea încercărilor va începe de la 0):  $h: A \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$

#### *Operația de inserare*

Pentru a insera elementul *x* în tabela *T* se examinează, în ordine, pozиțiile  $h(x, 0)$ ,  $h(x, 1)$ , ...,  $h(x, m-1)$ . Această secvență de pozиții trebuie să fie o permutare a mulțimii  $\{0, 1, \dots, m-1\}$ .

```
Insert (T, x)
{ i=0;
  do {j=h(x, i)
      if (T[j]==NIL) {T[j]=k; return j;}
      else i++;
    }
  while (i<m);
  error ("hash table overflow");
  return -1; }
```

#### *Operația de căutare*

Căutarea unui element în tabelă presupune examinarea aceleiași secvențe de poziții ca și inserarea. Dacă în tabel nu se efectuează ștergeri, căutarea se termină fără succes la întâlnirea primei poziții libere.

```
Search (T, x)
{ i=0;
  do
    {j=h(x, i);
     if (T[j]==x) return j;
     i++; }
  while (T[j]!=NIL || i<m);
  return NIL;
```

#### *Operația de ștergere*

Ștergerea unui element dintr-o tabelă *hash* este dificilă. Nu putem doar marca poziția *i* pe care este plasată cheia *x* ca fiind liberă. În acest mod, căutarea oricărei chei pentru care pe parcursul inserării am examinat poziția *i* și am găsit-o ocupată se va termina fără succes.

O soluție ar fi ca la ștergerea unui element, pe poziția respectivă să nu plasăm valoarea *NIL*, ci o altă valoare specială (pe care o vom denumi *DELETED*). Funcția de inserare poate fi modificată astfel încât să insereze elementul *x* pe prima poziție pe care se află *NIL* sau *DELETED*.

Dacă alegem această soluție, timpul de execuție al căutării nu mai este proporțional cu factorul de încărcare a tabelei.

Din acest motiv, tehnica de rezolvare a coliziunilor prin înlănțuire este mai utilizată pentru cazul în care structura de date suportă și operații de ștergere.

#### *Observație*

Pentru ca distribuția cheilor să fie uniformă, funcția *hash* trebuie să genereze cu probabilitate egală oricare dintre cele  $m!$  permutări ale mulțimii  $\{1, 2, \dots, m-1\}$ .

Pentru a determina secvența pozиțiilor de examinat se utilizează trei tehnici. Fiecare dintre aceste trei tehnici garantează că, pentru orice cheie, secvența pozиțiilor de examinat este o permutare a mulțimii  $\{0, 1, \dots, m-1\}$ . Nici una dintre acestea nu asigură o distribuție perfect uniformă.

#### *Examinare liniară*

Considerând o funcție *hash* „obișnuită”,  $h': A \rightarrow \{0, 1, \dots, m-1\}$  (denumită funcție *hash auxiliară*), tehnica examinării liniare utilizează funcția:

$$h(x, i) = (h'(x) + i) \% m$$

#### *Observații*

Deoarece prima poziție din secvență determină întreaga secvență de pozиții de examinat, deducem că există doar  $m$  secvențe de examinare distincte.

Examinarea liniară este ușor de implementat, dar generează un fenomen denumit *primary clustering* (secvența de pozиții ocupate devine mare și astfel timpul mediu de execuție al căutării crește).

**Examinare pătratică**

Examinarea pătratică utilizează o funcție *hash* de forma:

$$h(x, i) = (h'(x) + c_1 i + c_2 i^2) \% m,$$

unde  $h'$  este funcția *hash* auxiliară, iar  $c_1$  și  $c_2 \neq 0$  sunt constante auxiliare.

**Observații**

Această metodă funcționează mai bine decât examinarea liniară, dar, pentru a garanta faptul că întreaga tabelă este utilizată,  $m$ ,  $c_1$  și  $c_2$  trebuie să îndeplinească anumite condiții.

Vom considera că  $m$  este o putere a lui 2. O variantă bună de examinare pătratică este:

```
i=h'(x); j=0;
do
    if (T[i]==NIL)
        T[i]=x; return i;
    j=(j+1)%m; i=(i+j)%m;
}
while(1);
```

**Hashing dublu**

Hashing-ul dublu utilizează o funcție de forma

$$h(x, i) = (h_1(x) + i h_2(x)) \% m,$$

unde  $h_1$  și  $h_2$  sunt funcții *hash* auxiliare.

Pentru ca întreaga tabelă să fie examinată, valoarea  $h_2(x)$  trebuie să fie relativ primă cu  $m$ . O metodă sigură este de a-l alege pe  $m$  ca o putere a lui 2, iar  $h_2$  să producă numai valori impare. Sau  $m$  să fie un număr prim, iar  $h_2$  să producă numere  $\leq m$ .

**Exemplu**

$$\begin{aligned} h_1(x) &= x \% m; \\ h_2(x) &= 1 + (x \% m) \end{aligned}$$

unde  $m$  este  $\leq m$  (de exemplu,  $m=1$ ).

**Observație**

Hashing-ul dublu este mai eficient decât cel liniar sau pătratic.

**5.4. Hashing perfect**

Există aplicații în care mulțimea cheilor nu se schimbă (de exemplu, cuvintele rezervate ale unui limbaj de programare; numele fișierelor scrise pe un CD etc.). Hashing-ul poate fi utilizat și pentru probleme în care mulțimea cheilor este statică (odată ce au fost inserate în tabela *hash*, aceasta nu se mai schimbă). În acest caz, timpul de execuție în cazul cel mai defavorabil este  $O(1)$ .

Hashing-ul se numește *hashing perfect* dacă nu există coliziuni.

Hashing-ul perfect se realizează pe două niveluri. La primul nivel este similar cu hashing-ul cu înlățuire: cele  $n$  chei vor fi distribuite cu ajutorul unei funcții *hash*  $h$

în  $m$  locații. Dar în loc de a crea o listă înlățuită cu toate cheile distribuite în locația  $i$ , la cel de-al doilea nivel vom utiliza căte o tabelă *hash* secundară,  $S_i$ , care are asociată o funcție *hash*  $h_i$  pentru fiecare locație în care există coliziuni.

Pentru a garanta faptul că nu vor exista coliziuni în tabela *hash* secundară, dimensiunea  $m_i$  a talelei *hash* secundare  $S_i$  trebuie să fie cel puțin egală cu  $n_i^2$  (unde  $n_i$  este numărul de chei care sunt distribuite în locația  $i$ ).

Evident, descrierea grupurilor de la primul nivel trebuie să conțină și funcția *hash* aleasă pentru grupul respectiv, precum și dimensiunea spațiului de memorie alocat talelei *hash* secundare.

O alegere a funcției *hash*  $h$  dintr-o familie universală de funcții *hash* asigură că dimensiunea totală a spațiului de memorie utilizat este  $O(n)$ .

**5.5. Algoritmul Rabin-Karp**

Fie  $T$  un sir de  $n$  caractere și  $P$  un pattern de  $m$  caractere. Să se verifice dacă  $P$  apare sau nu ca subsăvență în sirul  $T$ .

**Soluție**

Michael Rabin și Richard Karp au conceput un algoritm de *pattern-matching* bazat pe *hashing*. Ideea este că dacă două siruri au aceeași valoare *hash*, atunci ar putea să coincidă; în caz contrar, sigur sunt diferite. Cu alte cuvinte, răspunsul NU este întotdeauna corect, iar răspunsul DA este corect cu o probabilitate mare.

Algoritmul preprocesează patternul în  $O(m)$ . Căutarea se execută în cazul cel mai defavorabil în  $O((n-m+1)m)$ , dar, în medie, timpul de execuție este mai bun.

Să notăm cu  $d$  numărul de caractere distincte care apar în text. Un sir de lungime  $m$  poate fi considerat un număr în baza  $d$  având  $m$  cifre.

În continuare, vom nota cu  $p$  valoarea în baza 10 asociată patternului  $P$ , iar cu  $t_k$  valoarea în baza 10 a subsecvenței din  $T$  care începe la poziția  $k$  ( $T[k..k+m-1]$ ). Evident,  $p=t_k$ , dacă și numai dacă  $P=T[k..k+m-1]$ .

Valorile  $t_k$ , pentru  $k=0, \dots, n-m+1$  se pot calcula în  $O(n-m+1)$ , deoarece  $t_{k+1}$  se poate determina din  $t_k$ , utilizând următoarea formulă:

$$t_{k+1} = d * (t_k - d^{m-1} * T[k]) + T[k+m].$$

Singura dificultate constă în faptul că  $p$  și  $t_k$  pot fi numere foarte mari. Din această cauză, operațiile cu ele nu se execută în timp constant.

Pentru a rezolva această problemă vom lucra cu  $p$  și  $t_k$  modulo  $q$ , unde  $q$  este un număr prim convenabil ales (astfel încât  $d^m < MAXLONGINT$ ), deci vom utiliza o funcție *hash*. Problema este că  $p \% q = t_k \% q$  nu implică faptul că  $p = t_k$  (adică apar coliziuni). Dar dacă  $p \% q != t_k \% q$ , atunci sigur  $P != T[k..k+m-1]$ . Deci putem utiliza testul modulo  $q$  ca o euristică, urmând să testăm egalitatea dintre  $P$  și  $T[k..k+m]$ .

**Exemplu**

Fie  $P="31415"$  și  $T="2359023141526739921"$ . În acest caz  $d=10$  (sunt doar cele 10 cifre),  $q=13$ ,  $m=|P|=5$  ( $P$  are 5 caractere) și  $n=|T|=19$  ( $T$  are 19 caractere).

Valoarea p a lui P este  $p=31415926531=7$

Valorile funcției hashing pentru subsecvențele de lungime 5 ale textului:

$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$	$t_{10}$	$t_{11}$	$t_{12}$	$t_{13}$	$t_{14}$
8	9	3	11	0	1	7	8	4	5	10	11	7	9	11

Se observă că  $p=t_6$  și  $p=t_{12}$ . Prima este potrivirea corectă, în schimb a două este o potrivire falsă.

```
RABIN_KARP (T, P, d, q)
{n=strlen(T); m=strlen(P);
h=dm-1%q;
p=0; t0=0;
for (i=0; i<m; i++) //preprocesare pattern
{ p=(d*p+P[i])%q;
t0=(d*t0+T[i])%q;
}
for (k=0; k<=n-m; k++)
{if (p==t0)
    if (P==T[k..k+m-1])
        printf("YES"); return;
    t0=(t0+d*q-T[k]*h)%q;
    t0=(t0*d+T[k+m])%q;
}
printf("NO"); return;
}
```

Desigur, la scrierea aplicației trebuie implementate funcții pentru calculul lui h și pentru verificarea egalității  $P==T[k..k+m-1]$ . De asemenea, în funcție de mulțimea caracterelor care intervin în datele de prelucrat, trebuie alese în mod corespunzător d și q. Pentru caracterele din codul ASCII se poate alege d=128 și q=131. Un exemplu de implementare poate fi următorul:

```
#include <stdio.h>
#include <string.h>
#define NMAX 1000

char T[NMAX], P[NMAX];
int n, m;

unsigned long putere(int d, int m)
{ int i;
  unsigned long p=1;
  for (i=1; i<m; i++) p*=d;
  return p;
}

int EQ(char *P, char *T, int k)
{ int i;
  for (i=0; i<m; i++)
    if(P[i]!=T[k+i]) return 0;
  return 1;
}
```

```
int RABIN_KARP (char *T, char *P, int d, int q)
{ unsigned long h, p, t0;
int i, k;
n=strlen(T); m=strlen(P);
h=putere(d, m)%q;
p=0; t0=0;
for (i=0; i<m; i++) //calculez p si t0 initial
{ p=(d*p+P[i])%q; t0=(d*t0+T[i])%q; }
for (k=0; k<=n-m; k++)
{ if (p==t0) //incerc toate posibilele deplasamente
    if (EQ(P, T, k)) return k; //am gasit la pozitia k
    t0=(t0+d*q-T[k]*h)%q; //recalculez t0
    t0=(t0*d+T[k+m])%q;
}
return -1; //nu am gasit
}

int main()
{ int poz;
freopen("rk_sir.in", "r", stdin);
freopen("rk_sir.out", "w", stdout);
scanf("%s%s", T, P);
poz=RABIN_KARP(T, P, 128, 131); //d=128, q=131
if (poz>=0) printf("Gasit in pozitia %d\n", poz);
else printf("Subsir negasit\n");
fclose(stdout);
return 0;
}
```

## 5.6. Probleme propuse

### 1. Broasca buclucașă

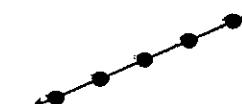
În Coreea, obrăznicia broscuței *cheonggaeguri* este legendară. Reputația este bineînerâtă, deoarece aceste broscuțe săr noaptea prin plantațiile de orez, strivind plantele de orez. Dimineață, după ce ați observat care plante au fost strivite, vreți să identificați drumul broscuței care a produs cele mai mari stricăciuni. O broscuță sare pe plantație în linie dreaptă, făcând salturi de lungimi egale:



Broscuțe diferite pot face salturi:  
de lungimi diferite



și  
în direcții diferite



Plantația de orez are plantele plasate în punctele de intersecție ale unui caroaj, ca în figura 1, și broscuțele buclucașe traversează plantația, intrând din exterior prin una dintre laturi și ieșind la exterior pe o altă latură, ca în figura 2:

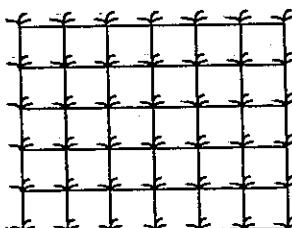


Figura 1

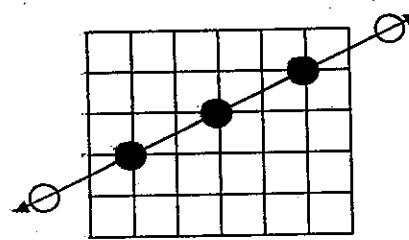


Figura 2

Mai multe broscuțe pot topăi pe plantație, sărind de la o plantă de orez la alta. La orice salt broscuța aterizează pe o plantă și o strivete, ca în figura 3. Remarcați că unele plante pot ateriza mai multe broscuțe în cursul nopții. Bineînțeles, voi nu puteți vedea liniile care marchează traseele broscuțelor sau salturile pe care acestea le fac în afara plantației – pentru situația ilustrată în figura 3, ceea ce puteți vedea este ilustrat în figura 4:

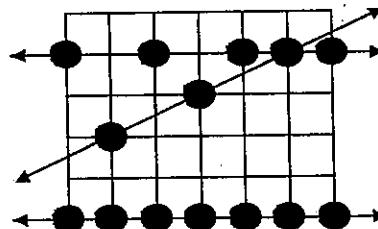


Figura 3

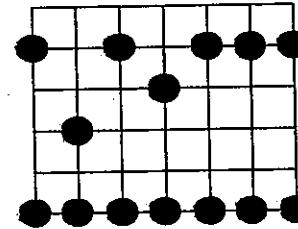


Figura 4

Din figura 4 puteți reconstitui toate traseele pe care broscuțele le-ar fi putut urma pentru a traversa plantația. Vă interesează numai broscuțele care au aterizat pe cel puțin trei plante de orez pe traseul lor de pe plantație. Un astfel de traseu este denumit traseu-de-broască. În acest caz, deducem că cele trei trasee ilustrate în figura 3 sunt trasee-de-broască (există și alte trasee-de-broască posibile). Traseul vertical de la coloana 1 în jos poate fi un traseu cu saltul de lungime 4, dar nu e traseu-de-broască, deoarece conține numai două plante strivite; traseul diagonal, care include plantele de pe linia 2 coloana 3, linia 3 coloana 4, și linia 6 coloana 7 conține trei plante strivite, dar nu putem stabili o lungime pentru salt astfel încât traseul să conțină cel puțin 3 plante la distanțe egale, prin urmare, nu este un traseu-de-broască. Observați că pe linia determinată de un traseu-de-broască pot exista și alte plante strivite, dar pe care nu s-a aterizat în mod necesar pe acest traseu (vezi planta din poziția (2, 6) de pe traseul orizontal de-a lungul liniei 2 din figura 4), și de fapt, unele plante strivite pot să nu aparțină nici unui traseu-de-broască.

### Cerință

Scrieți un program care să determine numărul de salturi de pe un traseu-de-broască care dintre toate traseele-de-broască posibile a aterizat pe un număr maxim de plante (și astfel a produs cea mai mare pagubă în recolta de orez). În figura 4 acesta ar fi traseul care merge de-a lungul liniei 6, producând rezultatul 7.

### Date de intrare

Programul va citi de la intrarea standard. Prima linie conține două numere întregi R și C, reprezentând numărul de linii, respectiv numărul de coloane din plantația de orez,  $1 \leq R, C \leq 5000$ . Cea de-a doua linie conține un singur număr întreg N, reprezentând numărul de plante de orez strivite,  $3 \leq N \leq 5000$ . Fiecare dintre următoarele N linii conține câte două numere întregi separate prin spațiu, reprezentând numărul liniei ( $1 \leq \text{numărul liniei} \leq R$ ) și numărul coloanei ( $1 \leq \text{numărul coloanei} \leq C$ ) unei plante strivite. Fiecare plantă strivită apare o singură dată.

### Date de ieșire

Programul va scrie la ieșirea standard o linie cu un singur număr întreg reprezentând numărul de plante strivite de pe traseul-de-broască care a produs cele mai mari pagube, dacă există cel puțin un traseu-de-broască; altfel, va scrie 0.

### Exemple

input	output	input	output
6 7	7	6 7	4
14		18	
2 1		1 1	
6 6		6 2	
4 2		3 5	
2 5		1 5	
2 6		4 7	
2 7		1 2	
3 4		1 4	
6 1		1 6	
6 2		1 7	
2 3		2 1	
6 3		2 3	
6 4		2 6	
6 5		4 2	
6 7		4 4	
		5 4	
		5 5	
		6 6	

### Explicații

Exemplul 1 corespunde figurii 4.  
Exemplul 2 corespunde figurii 5.

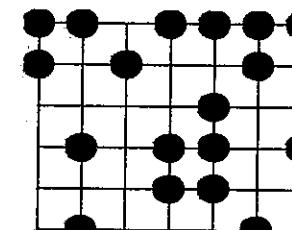
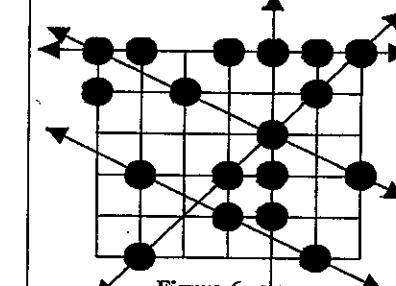


Figura 5



Numărul maxim de plante strivite de o broască este 4.

**2. a007**

Agentul 007 are de distrus o tabără de teroriști. Tabără de teroriști este formată din mai multe obiective (depozite de muniție, pavilioane pentru teroriști etc.), considerate punctiforme în plan. Agentul 007 primește de la serviciul de informații o hartă cu  $n$  obiective din tabără teroriștilor, date prin coordonatele carteziene. Pe lângă hartă, agentul 007 mai primește și o armă specială (construită pentru această misiune). Arma primită are două tevi și permite tragerea simultană pe aceeași direcție (rectilinie), dar în sens invers, a două rachete cu aceeași viteză. După ce se trage cu arma, odată cu atingerea unei ținte explodează și cealaltă rachetă (chiar dacă aceasta din urmă nu și-a atins țintă). Agentul 007 vrea să distrugă tabără cât mai repede și cu cât mai puține rachete; pentru acest lucru, el studiază posibilitatea să se așeze într-un punct din tabără (diferit de obiective) care să permită trageri eficace, adică la fiecare tragere să distrugă câte două obiective simultan. Determinați dacă este posibil să se găsească un astfel de punct.

*Date de intrare*

În fișierul de intrare a007.in, pe prima linie se află numărul de teste  $k$ , după care urmează date pentru fiecare test. Pentru fiecare test, pe o linie se află  $n$ , iar pe următoarele  $n$  linii sunt coordonatele obiectivelor din tabără teroriștilor (separate printr-un spațiu în ordinea abscisă ordonată).

*Date de ieșire*

În fișierul a007.out se vor scrie  $k$  linii, pe fiecare linie se va scrie 1 dacă există soluție și 0 dacă nu există soluție. În cazul în care există soluție se vor scrie în continuare pe aceeași linie, separate printr-un spațiu, coordonatele punctului cerut (numere reale trunchiate la 4 zecimale, în ordinea abscisă ordonată).

*Restricții*

$0 \leq n \leq 10000$

$1 \leq k \leq 3$

Coordonatele punctelor sunt întregi din intervalul  $[-10000, 10000]$ .

*Exemplu*

a007.in

2

4

10 0

10 10

0 10

0 0

6

0 0

10 0

2 10

12 0

5 0

7 0

a007.out

1 5.0000 5.0000

0

## 6. Geometrie computațională

### 6.1. Introducere

Geometria computațională se ocupă cu studierea algoritmilor pentru rezolvarea problemelor de geometrie. Problemele de geometrie se împart în două mari categorii: geometrie în plan (2D) și geometrie în spațiu (3D). În acest capitol ne propunem să studiem câteva aspecte de bază ale geometriei computaționale în plan.

Pentru o problemă de geometrie, datele de intrare sunt de obicei mulțimi de puncte, mulțimi de segmente, mulțimi de vârfuri ale unui poligon etc. Datele de ieșire reprezintă uneori un răspuns la o întrebare despre datele de ieșire (de exemplu dacă două segmente se intersecțează sau nu) sau pot reprezenta o nouă mulțime (de exemplu, mulțimea punctelor, dintre cele inițiale, care sunt vârfurile poligonului convex de arie minimă ce conține punctele date – înfășurătoarea convexă).

În general, un *punct* în plan este identificat prin cele două coordonate ale sale referitoare la un sistem de coordonate cartezian (abscisa și ordinata –  $x$  și  $y$ ). Prin urmare, vom nota un punct de coordonate  $x$ ,  $y$  sub forma  $P(x, y)$ . În funcție de problemă, coordonatele unui punct pot fi întregi sau reale.

Vom descrie un punct ca o structură cu două câmpuri  $x$  și  $y$  reprezentând cele două coordonate ale sale, un segment de dreaptă ca o structură cu două câmpuri de tip *Punct*, reprezentând extremitățile sale.

```
typedef struct { double x, y; } Punct;
typedef struct {Punct A, B;} Segment;
Punct P, P1, P2;
Segment L;
```

### 6.2. Puncte și drepte

Fiind date două puncte distincte în plan,  $P_1(x_1, y_1)$  și  $P_2(x_2, y_2)$ , vom nota cu  $P_1P_2$  segmentul de dreaptă care are o extremitate în  $P_1$  și o extremitate în  $P_2$ . De multe ori este importantă și ordinea punctelor, caz în care se vorbește despre segmentul *orientat*  $P_1P_2$ .

Distanța dintre punctele  $P_1$  și  $P_2$  este dată de formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

unde prin *sqrt* am notat radicalul de ordinul 2.

### Pozitia unui punct față de un segment. Coliniaritate

Prima problemă care se pune este de a determina dacă un punct oarecare  $P$  aparține segmentului  $P_1P_2$ , iar în caz negativ, care este poziția sa față de dreapta-suport a segmentului  $P_1P_2$ , considerând de data aceasta segmentul *orientat*  $\overrightarrow{P_1P_2}$ .

Pentru a testa dacă punctul  $P$  aparține dreptei determinate de punctele  $P_1$  și  $P_2$ , se scrie ecuația dreptei care trece prin punctele  $P_1$  și  $P_2$  și se testează dacă ecuația este verificată de punctul  $P$ . Ecuația dreptei poate fi scrisă sub forma:

$$(y - y_1) * (x_2 - x_1) - (x - x_1) * (y_2 - y_1) = 0.$$

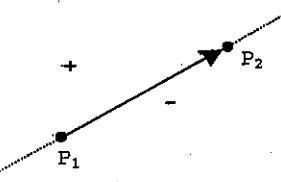
Verificarea ecuației dreptei nu este suficientă, deoarece orice punct de pe dreapta ce trece prin  $P_1$  și  $P_2$  verifică ecuația dreptei. Pentru a testa dacă  $P$  aparține segmentului  $P_1P_2$  mai trebuie verificat ca punctul  $P$  să fie în interiorul dreptunghiului ce are ca diagonală segmentul  $P_1P_2$  sau, în cazul în care segmentul este paralel cu una dintre axe, în interiorul segmentului, adică:

$$x \geq \min(x_1, x_2) \text{ și } x \leq \max(x_1, x_2) \text{ și/sau}$$

$$y \geq \min(y_1, y_2) \text{ și } y \leq \max(y_1, y_2).$$

Rezolvarea celei de-a doua cerințe utilizează rezultatul verificării ecuației dreptei de către coordonatele punctului  $P$ .

În cazul în care punctul  $P$  nu se află pe dreapta-suport ce conține segmentul orientat  $\overrightarrow{P_1P_2}$ , prin înlocuirea valorilor  $x$  și  $y$  în ecuația dreptei se obține o valoare reală nenulă. Dreapta-suport împarte planul în două regiuni (semiplane), să le spunem „stânga” și „dreapta”. Pentru toate punctele aflate în „stânga” dreptei-suport valoarea obținută este strict pozitivă, iar pentru toate punctele aflate în „dreapta” valoarea obținută este strict negativă.



Funcția `verifica()` calculează valoarea funcției liniare care descrie dreapta determinată de  $P_1$  și  $P_2$  în punctul  $P$ . Funcția `verifica()` returnează valoarea 0 dacă și numai dacă punctele  $P_1$ ,  $P_2$ ,  $P$  sunt coliniare. Funcția `apartine()` verifică dacă punctul  $P$  aparține segmentului  $P_1P_2$ .

```
int verifica(Punct P, Punct P1, Punct P2)
{return (P.y-P1.y)*(P2.x-P1.x)-(P.x-P1.x)*(P2.y-P1.y); }

int apartine(Punct P, Punct P1, Punct P2)
{if (P1.x==P2.x) //dreapta verticala
    if (P.y>=min(P1.y, P2.y) && P.y<=max(P1.y, P2.y))
        return 1; //in segment
    else return 0;
if (P.x>=min(P1.x, P2.x) && P.x<=max(P1.x, P2.x))
    return 1; //in segment
return 0; }
```

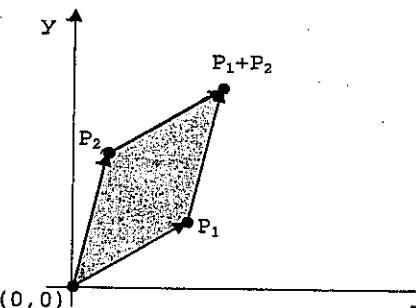
### Pozitia segmentului orientat $\overrightarrow{P_0P_1}$ față de segmentul orientat $\overrightarrow{P_0P_2}$

Pentru început vom considera că cele două segmente orientate au capătul comun originea axelor,  $P_0(0, 0)$ , și vom defini noțiunea de *produs încrucișat*.

Produsul încrucișat  $P_1 \times P_2$  se definește astfel:

$$P_1 \times P_2 = x_1 * y_2 - x_2 * y_1 = \det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix}$$

Din punct de vedere geometric, produsul încrucișat  $P_1 \times P_2$  reprezintă aria *cu semn* a paralelogramului format din punctele  $(0, 0)$ ,  $P_1(x_1, y_1)$ ,  $P_2(x_2, y_2)$  și  $P_1 + P_2(x_1 + x_2, y_1 + y_2)$ .



Acum, dacă valoarea produsului încrucișat este pozitivă, atunci punctul  $P_1$  este poziționat în sensul acelor de ceasornic față de  $P_2$ , iar, dacă acest produs este negativ,  $P_1$  este poziționat în sens trigonometric față de  $P_2$ . Dacă produsul este 0, atunci cele două segmente sunt coliniare.

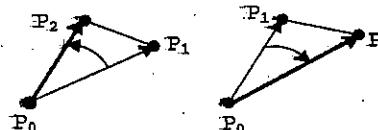
Să considerăm acum că segmentele orientate  $\overrightarrow{P_0P_1}$  și  $\overrightarrow{P_0P_2}$  au originea comună  $P_0$  oarecare, diferită de originea axelor de coordonate. Pentru a putea aplica cele descrise mai sus este suficient să translatăm cele două segmente orientate astfel încât ambele să aibă ca extremitate inițială originea axelor de coordonate. Cu alte cuvinte va trebui să calculăm produsul:

$$(P_1 - P_0) * (P_2 - P_0) = (x_1 - x_0) * (y_2 - y_0) - (x_2 - x_0) * (y_1 - y_0).$$

Dacă valoarea produsului este pozitivă, atunci  $\overrightarrow{P_0P_1}$  este poziționat în sensul acelor de ceasornic față de  $\overrightarrow{P_0P_2}$ , iar, dacă acest produs este negativ,  $\overrightarrow{P_0P_1}$  este poziționat în sens trigonometric față de  $\overrightarrow{P_0P_2}$ .

### Pozitia a două segmente consecutive

Date două segmente de dreaptă  $P_0P_1$  și  $P_1P_2$ , care au punctul  $P_1$  comun, se pune întrebarea dacă, „trecând” de la punctul  $P_0$  la  $P_1$  și apoi la  $P_2$ , la trecerea de pe un segment pe altul în punctul  $P_1$ , trecerea se face în sensul acelor de ceasornic (întoarcere la dreapta) sau în sens trigonometric (întoarcere la stânga).



Vom folosi și de data aceasta produsul încrucișat și rezultatele obținute în paragraful anterior. Astfel, se verifică poziția segmentului orientat  $P_0P_2$  față de segmentul orientat  $P_0P_1$  prin calcularea produsului  $(P_1 - P_0) \cdot (P_2 - P_0)$ . Dacă acest produs este negativ, atunci în  $P_1$  se va face o întoarcere în sensul acelor de ceasornic spre  $P_2$ . Dacă produsul este pozitiv, în  $P_1$  se va face o întoarcere în sens trigonometric spre  $P_2$ . În fine, dacă produsul este 0, cele trei puncte sunt coliniare.

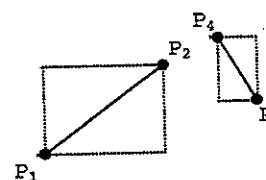
Funcția `ceas_trig()` returnează o valoare strict pozitivă dacă  $P_0P_1P_2$  reprezintă o întoarcere la stânga, o valoare strict negativă dacă  $P_0P_1P_2$  reprezintă o întoarcere la dreapta, respectiv valoarea 0 dacă  $P_0, P_1, P_2$  sunt coliniare.

```
int ceas_trig(Punct P0, Punct P1, Punct P2)
{
    return (P1.x - P0.x) * (P2.y - P0.y) - (P2.x - P0.x) * (P1.y - P0.y);
}
```

### Intersecția a două segmente de dreaptă

Pentru a determina dacă două segmente de dreaptă  $P_1P_2$  și  $P_3P_4$  se intersectează sau nu vom proceda în două etape :

1. *Testul de respingere rapidă*. Dacă dreptunghiul cu diagonală  $P_1P_2$  nu se intersectează cu dreptunghiul cu diagonală  $P_3P_4$ , evident că nici segmentul  $P_1P_2$  nu se intersectează cu  $P_3P_4$ .



Pentru aceasta este suficient să nu fie îndeplinită condiția compusă :

$$x2 \geq x3 \quad \& \quad x4 \geq x1 \quad \& \quad y2 \geq y3 \quad \& \quad y4 \geq y1,$$

unde :

$$\begin{aligned} x1 &= \min(P1.x, P2.x); \quad x2 = \max(P1.x, P2.x); \\ y1 &= \min(P1.y, P2.y); \quad y2 = \max(P1.y, P2.y); \\ x3 &= \min(P3.x, P4.x); \quad x4 = \max(P3.x, P4.x); \\ y3 &= \min(P3.y, P4.y); \quad y4 = \max(P3.y, P4.y); \end{aligned}$$

2. *Verificarea intersecției*. Dacă cele două segmente „trec” de testul de respingere rapidă, este posibil ca ele să se intersecteze. Pentru ca ele să se intersecteze este obligatoriu ca fiecare dintre cele două segmente să intersecteze dreapta-suporț care conține celălalt segment.

Segmentul  $P_1P_2$  intersectează dreapta ce conține segmentul  $P_3P_4$  dacă punctul  $P_1$  se află de o parte a dreptei și punctul  $P_2$  se află de cealaltă parte a dreptei. În mod analog, segmentul  $P_3P_4$  intersectează dreapta ce conține segmentul  $P_1P_2$  dacă punctul  $P_3$  se află de o parte a dreptei, iar punctul  $P_4$  se află de cealaltă parte a dreptei. Pentru a verifica cele două condiții vom utiliza funcția care indică poziția a două segmente consecutive, studiată în paragraful precedent.

Am văzut că funcția returnează o valoare negativă pentru sens orar, 0 pentru puncte coliniare și o valoare pozitivă pentru sens trigonometric. Atunci, dacă produsul dintre rezultatul apelului funcției pentru punctele  $P_1, P_2, P_3$  și rezultatul apelului funcției pentru punctele  $P_1, P_2, P_4$  este negativ, înseamnă că punctul  $P_3$  se află de o parte a segmentului  $P_1P_2$ , iar punctul  $P_4$  se află de cealaltă parte. În mod similar, dacă produsul dintre rezultatul apelului funcției pentru punctele  $P_3, P_4, P_1$  și rezultatul apelului funcției pentru punctele  $P_3, P_4, P_2$  este negativ, înseamnă că punctul  $P_1$  se află de o parte a segmentului  $P_3P_4$  iar punctul  $P_2$  se află de cealaltă parte. Dacă produsul este 0 înseamnă că unul dintre capetele segmentului se află pe celălalt segment.

```
int respingere_rapida(Punct P1, Punct P2, Punct P3, Punct P4)
{
    int x1, x2, y1, y2, x3, x4, y3, y4;
    x1=min(P1.x, P2.x); x2=max(P1.x, P2.x);
    y1=min(P1.y, P2.y); y2=max(P1.y, P2.y);
    x3=min(P3.x, P4.x); x4=max(P3.x, P4.x);
    y3=min(P3.y, P4.y); y4=max(P3.y, P4.y);
    return !(x2>=x3 && x4>=x1 && y2>=y3 && y4>=y1);
}

int verific_intersecție(Punct P1, Punct P2, Punct P3, Punct P4)
{
    if (ceas_trig(P1, P2, P3)*ceas_trig(P1, P2, P4)>0) return 0;
    if (ceas_trig(P3, P4, P1)*ceas_trig(P3, P4, P2)>0) return 0;
    return 1;
}
```

### 6.3. Poligoane

Un poligon este determinat de o secvență de puncte în plan (denumite vîrfuri ale poligonului). Oricare două puncte consecutive în secvență determină o latură a poligonului (se consideră că după ultimul punct din secvență urmează primul).

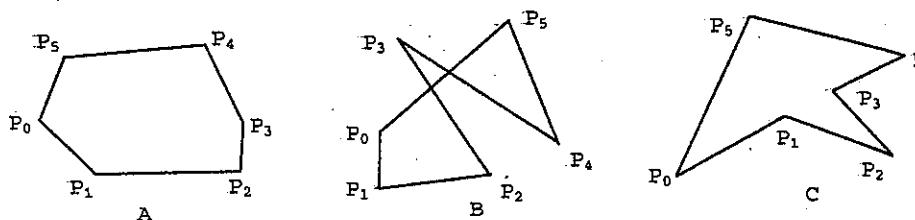
Prin urmare, reprezentăm un poligon ca un tablou cu elemente de tip `Punct`, în care reținem, în ordine, vîrfurile poligonului.

```
#define DIM 200
Punct P[DIM];
int n;
```

Dacă poligonul definit are  $n$  vîrfuri  $P_0, P_1, \dots, P_{n-1}$ , atunci laturile lui vor fi  $P_0P_1, P_1P_2, \dots, P_{n-2}P_{n-1}, P_{n-1}P_0$ .

*Exemple*

În figurile următoare sunt ilustrate trei poligoane cu  $n=6$  vârfuri.



Observați că în poligonul B există laturi care se intersectează (în alt punct decât vârfurile poligonului).

Un poligon se numește *simple* dacă nu se autointersectează (nu există două laturi ale poligonului care să se intersecteze în alt punct decât vârfurile poligonului).

În cele ce urmează vom discuta numai despre poligoane *simple*. Poligoanele simple se împart și ele în două categorii: poligoane *convexe* și poligoane *neconvexe*.

Un poligon se numește *convex* dacă pentru orice latură a sa, vârfurile poligonului se află de aceeași parte a dreptei-suport a laturii.

Observați în figură că poligonul A este convex, iar poligonul B este neconvex.

Laturile poligonului împart mulțimea punctelor planului în două submulțimi: punctele *interioare* poligonului și punctele *exteroare* poligonului. Punctele aflate pe laturile poligonului se consideră ca aparținând interiorului acestuia.

Oricare ar fi două puncte aparținând interiorului unui poligon convex, segmentul determinat de cele două puncte este integral inclus în interiorul poligonului.

*Aria unui triunghi*

Un triunghi este un poligon cu trei laturi. El poate fi identificat prin coordonatele celor trei vârfuri ale sale  $P_0(x_0, y_0)$ ,  $P_1(x_1, y_1)$  și  $P_2(x_2, y_2)$ .

Lungimile laturilor triunghiului se calculează ca distanța dintre două puncte folosind formula:

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, \text{ pentru } 0 \leq j < i \leq 2.$$

Aria triunghiului se poate calcula folosind formula lui Heron. Pentru aceasta se calculează mai întâi lungimile laturilor triunghiului  $a$ ,  $b$ ,  $c$  folosind formula de mai sus, apoi se utilizează formula:

$$A_3 = \sqrt{p(p-a)(p-b)(p-c)}, \text{ unde } p = (a+b+c)/2.$$

Observăm faptul că pentru a calcula aria triunghiului utilizând această formulă trebuie să folosim atât operația de împărțire, cât și operația de extragere de radical, operații costisoare din punctul de vedere al timpului de execuție.

Există și o altă metodă, mai eficientă, care utilizează direct coordonatele celor trei puncte. Așa cum am arătat, produsul încrucișat are ca interpretare geometrică aria paralelogramului determinat de două segmente. Dar orice triunghi poate fi „văzut”

ca jumătate de paralelogram. Deci prin calcularea produsului încrucișat am calculat dublul ariei unui triunghi. Desigur, aria se calculează cu semn, deci trebuie să avem în vedere acest lucru și, fie să avem grijă la ordinea de citire a celor trei puncte, fie să considerăm valoarea în modul.

Deci aria triunghiului poate fi determinată cu formula:

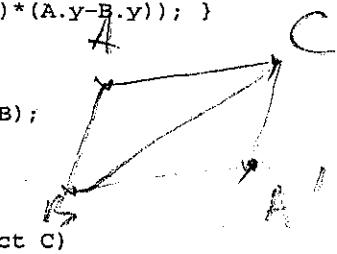
$$A_3 = |(x_1 - x_0) * (y_2 - y_0) - (x_2 - x_0) * (y_1 - y_0)| / 2.$$

```
double dist(Punct A, Punct B)
{ return sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y)); }
```

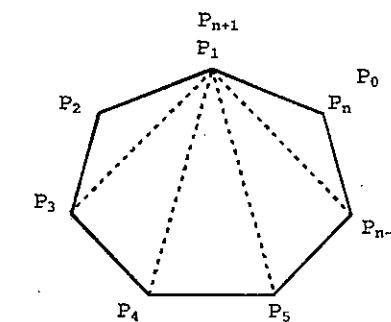
```
double Heron(Punct A, Punct B, Punct C)
```

```
{ double p, a, b, c;
  a=dist(B, C); b=dist(A, C); c=dist(A, B);
  p=(a+b+c)/2;
  return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

```
double Arie_Triunghi(Punct A, Punct B, Punct C)
{return abs((B.x-A.x)*(C.y-A.y)-(C.x-A.x)*(B.y-A.y))/2; }
```

*Aria unui poligon convex*

În cazul în care poligonul este convex, aria se calculează cu ușurință, partionând poligonul în triunghiuri, calculând aria fiecărui triunghi și însumând ariile obținute:



Pentru simplitate vom utiliza două componente suplimentare ( $P[0]$  și  $P[n+1]$ ), pe care le vom inițializa cu  $P[n]$ , respectiv  $P[1]$ :

```
double arie_convex (Poligon P, int n)
{ double arie=0;
  int i;
  for (i=2; i<n; i++)
    arie+=Arie_Triunghi(P[1], P[i], P[i+1]);
  return arie;
}
```

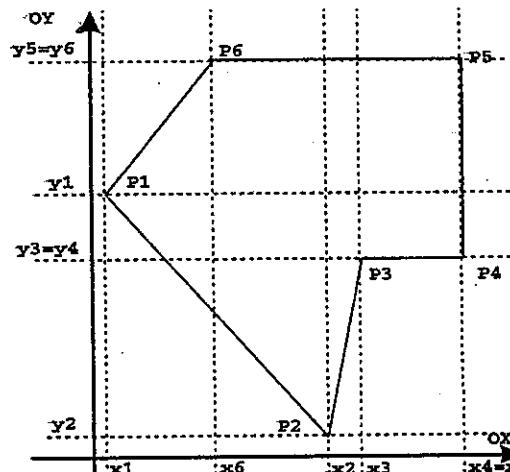
Observăm că pentru a determina aria unui poligon convex am utilizat funcția de determinare a ariei unui triunghi descrisă în secțiunea precedentă.

### Aria unui poligon oarecare

În cazul în care poligonul nu este în mod obligatoriu convex, calculul ariei se complică.

Pentru a deduce formula de calcul a ariei, vom trasa paralele la Oy prin fiecare vârf al poligonului. Oricare două drepte paralele consecutive, împreună cu latura poligonului corespunzătoare celor două vârfuri prin care sunt trasate și cu axa Ox, formează un trapez dreptunghic. Calculăm aria trapezului dreptunghic ca aria cu semn:

$$A_i = h_i \cdot (b_i + B_i) / 2 = (x_{i+1} - x_i) \cdot (y_{i+1} + y_i) / 2.$$



Calculând suma ariilor cu semn ale trapezelor dreptunghice în valoare absolută, obținem aria poligonului convex:

$$A = \left| \sum_{i=1}^n ((x_{i+1} - x_i) \cdot (y_{i+1} + y_i)) / 2 \right|$$

Reamintim că  $P_{n+1} = P_1$  și  $P_0 = P_n$ .

Pentru a calcula mai ușor această valoare, vom regrupa termenii, scoțând în factor pe  $x_i$ :

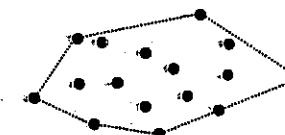
$$A = \left| \sum_{i=1}^n (x_i \cdot (y_{i+1} - y_{i-1})) / 2 \right|$$

```
double aria_oarecare(Polygon P, int n)
{ double aria=0;
  int i;
  for (i=1; i<=n; i++)
    aria+=P[i].x*(P[i+1].y-P[i-1].y);
  return abs(aria/2);
}
```

### Înfășurătoare convexă

Fie o mulțime de puncte în plan  $\{P_i, i=0, \dots, n-1\}$ . Se numește **înfășurătoare convexă** a mulțimii de puncte un poligon de arie minimă care are proprietatea că orice punct al mulțimii se găsește în interiorul lui.

Reamintim că am considerat că un punct care se află pe una dintre laturile poligonului aparține interiorului acestuia.



Un prim algoritm intuitiv folosește chiar definiția poligonului convex. Reamintim că un poligon este convex dacă pentru orice latură a sa mulțimea punctelor interioare poligonului se află de aceeași parte a dreptei-suport care conține latura respectivă. Tinând cont de definiție devine clar faptul că trebuie să determinăm acele segmente care au această proprietate. Deci pentru fiecare pereche de puncte alegem un punct care nu este pe segment, apoi verificăm dacă toate celelalte puncte sunt de aceeași parte a segmentului determinat de ele ca și punctul selectat și, în caz afirmativ, reținem perechea de puncte.

```
for (i=0; i<n-1; i++)
  for (j=i+1; j<n; j++)
  { OK=1;
    x=cauta(P[i], P[j]); //Px nu se află pe PiPj
    for (k=0; k<n && OK; k++)
      if (k!=i && k!=j)
        if (x*semn(P[i], P[j], P[k])<0) OK=0;
    if (OK) retine(P[i], P[j]);
  }
```

Acest algoritm are complexitatea timp  $O(n^3)$ , deoarece avem  $n(n-1)/2$  perechi de puncte care determină câte un segment și pentru fiecare dintre aceste segmente trebuie verificate toate  $n-2$  puncte rămase.

O altă metodă, denumită *scanarea Graham*, se bazează pe determinarea *unghiurilor polare* din jurul unui punct și ordonarea tuturor celorlalte puncte după unghiurile polare determinate.

Până acum am identificat un punct în plan prin cele două coordonate carteziene. O altă modalitate de a identifica un punct în plan este utilizarea *coordonatelor polare* (raza  $r$  și unghiul  $t$ ).

Raza  $r$  este distanța de la originea sistemului de coordonate la punctul  $P(x, y)$ :

$$r = \sqrt{x^2 + y^2}.$$

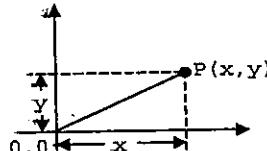
Unghiul  $t$  este unghiul pe care îl face segmentul  $OP$  cu axa Ox. Unghiul se determină din relațiile cunoscute în triunghiul dreptunghic:

$$t = \arctg(y/x).$$

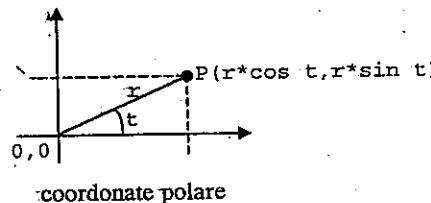
Se observă acum faptul că:

$$x = r \cdot \cos(t);$$

$$y = r \cdot \sin(t).$$

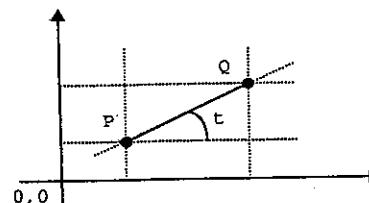


coordonate carteziene



coordonate polare

Pentru a utiliza aceste relații în problema noastră trebuie să definim de asemenea, ce înseamnă *unghi polar* al unui punct față de un alt punct. Unghiul polar pe care un punct  $P$  îl face cu un alt punct  $Q$  este determinat de dreapta orizontală care trece prin punctul  $P$  și dreapta-suport care conține segmentul  $PQ$ . Acest unghi poate fi determinat dacă se cunosc coordonatele carteziene ale punctelor  $P$  și  $Q$ , cu ajutorul relațiilor cunoscute în triunghiul dreptunghic. Observați că, practic, originea sistemului de coordonate este translată în punctul  $P$ .



Să revenim acum la problema determinării înfășurătorii convexe a unei mulțimi de puncte. Scanarea Graham determină înfășurătoarea convexă astfel:

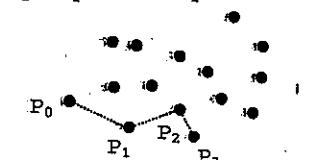
*Pasul 1:* Se determină un vârf extrem  $P_0$  care are coordonata  $y$  minimă, iar dacă mai sunt puncte cu această proprietate, cel cu coordonata  $x$  minimă dintre acestea. Acest pas se execută în timp liniar.

*Pasul 2:* Vom translata sistemul de coordonate, astfel încât originea să fie în punctul  $P_0$ . Se ordonează punctele în ordine crescătoare după unghiul polar. Dacă există mai multe puncte cu același unghi polar, se reține doar punctul cel mai depărtat de  $P_0$ . Pentru ordonare se utilizează un algoritm eficient cu ordin de complexitate  $O(n \log n)$ .

*Pasul 3:* Fie  $P_0, P_1, \dots, P_n$  punctele reînunțate în urma sortării. Se parcurg aceste puncte analizând pentru oricare trei puncte consecutive  $P_i, P_{i+1}, P_{i+2}$  dacă acestea reprezintă o întoarcere la stânga sau o întoarcere la dreapta. Dacă  $P_i, P_{i+1}, P_{i+2}$  reprezintă o întoarcere la dreapta, punctul  $P_{i+1}$  nu poate face parte din înfășurătoarea convexă, este eliminat și se continuă analiza cu tripletul  $P_{i-1}, P_i, P_{i+1}$ . Dacă  $P_i, P_{i+1}, P_{i+2}$  este o întoarcere la stânga, analiză continuă cu tripletul  $P_{i+1}, P_{i+2}, P_{i+3}$ .

### Exemplu

În figura următoare, la verificarea punctelor  $P_1, P_2, P_3$  în  $P_2$  se face o întoarcere spre dreapta, deci punctul  $P_2$  nu poate face parte din înfășurătoare.



Pentru verificare se utilizează o structură de date  $S$ , similară unei stive, diferență constând în faptul că se accesează primele două elemente din structură, nu doar primul element). Inițial se introduc în  $S$  primele două puncte  $P_0$  și  $P_1$ . Apoi se parcurg, pe rând, celelalte  $n-2$  puncte rămase. Pentru fiecare punct se consideră primele două elemente din  $S$ , al treilea fiind punctul respectiv. Dacă întoarcerea pentru aceste 3 puncte este la stânga, punctul se introduce în  $S$ . Dacă întoarcerea este spre dreapta înseamnă că punctul din vârf nu poate face parte din înfășurătoare, deci el este eliminat. Verificarea continuă până când se determină o secvență de trei puncte care reprezintă o întoarcere la stânga, apoi se trece la verificarea următorului punct.

Reamintim faptul că verificarea întoarcerii la stânga sau dreapta se face utilizând produsul încrucișat.

Pasul 3 are complexitatea timp  $O(n)$ , deoarece orice punct este considerat o singură dată. Deducem de aici faptul că timpul total este  $O(n \log n)$ , fiind determinat de timpul în care este realizată sortarea.

Funcțiile implementate sunt:

```

int varf;
Punct S[DIM];
void Graham()
{
    int i; double o;
    StangaJos(); //Determină punctul P0, y-minim, x-minim
    Sort(); //Sortează punctele după unghiul polar fata de P0
    //in P avem n elemente sortate P[0], P[1], ..., P[n-1]
    //pentru unghi egal sortat după distanța
    varf=0; PUSH(P[0]); PUSH(P[1]);
    for (i=2; i<n; i++)
    {
        o=Produs(S[varf-1], S[varf], P[i]);
        if (o==0) //coliniare
            {POP(); //elimin punct apropiat
             PUSH(P[i]); } //pun punct departat
        else
            if (o>0) PUSH(P[i]); //la stanga
            else
                //cat timp întoarcere la dreapta si se poate
                while (o<=0 && varf>1)
                    {POP(); //elimina varf si recalculeaza
                     o=Produs(S[varf-1], S[varf], P[i]); }
                PUSH(P[i]); }
    AfiseazaS();
}

```

```

void PUSH(Punct P)
{
    ++varf;
    S[varf]=P;
}

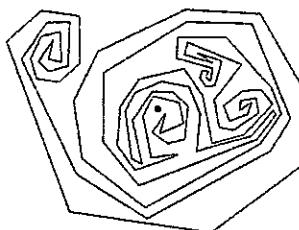
void POP()
{
    varf--;
}

int Produs(Punct P1, Punct P2, Punct P3)
{
    return (P2.x-P1.x)*(P3.y-P1.y)-(P3.x-P1.x)*(P2.y-P1.y);
}

```

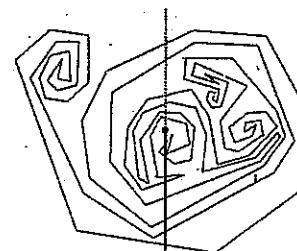
### Punct în poligon

În același mod în care s-a pus problema poziției unui punct față de o dreaptă se pune și problema unui punct față de un poligon: fiind dat un poligon  $P$  și un punct  $M$  în plan, unde este situat punctul respectiv? În interiorul poligonului, pe una dintre laturile lui sau în exteriorul poligonului?



Dacă poligonul este convex, problema nu este foarte complicată: de exemplu, putem verifica pentru fiecare latură dacă punctul  $M$  se află în același semiplan cu celelalte vârfuri. În caz afirmativ, punctul  $M$  se află în interiorul poligonului, în caz contrar se află în exterior. Această idee conduce la un algoritm liniar.

Pentru poligoane oarecare însă, problema nu este atât de simplă. Ideea de la care plecăm este următoarea: se trasează o dreaptă oarecare prin punctul  $M$ ; se alege o semidreaptă determinată de punctul  $M$  pe această dreaptă și se numără intersecțiile pe care semidreapta respectivă le are cu laturile poligonului. Fiecare intersecție semnifică trecerea din interiorul poligonului în exteriorul lui sau invers. Aceasta înseamnă că dacă numărul de intersecții este impar, punctul  $M$  se află în interiorul poligonului, iar dacă numărul de intersecții este par, punctul  $M$  se găsește în exteriorul poligonului. Pentru o implementare simplă vom duce o dreaptă paralelă cu Oy și vom lua în considerare semidreapta care „coboară” din  $M$ .



```

intersectii=0
pentru fiecare segment al poligonului
    dacă semidreapta care trece prin  $M(x,y)$  tăie segmentul
        intersectii++;
    dacă intersectii este impar Scrie „interior”;
        altfel Scrie „exterior”;

```

Implementarea codului solicită însă atenție la câteva detalii: ce se întâmplă când semidreapta care pornește din  $M$  intersectează poligonul chiar într-un vârf sau ce se întâmplă când punctul  $M$  se găsește pe o latură a poligonului?

În continuare ne vom baza pe ecuația parametrică a dreptei determinată de punctele  $P_1$  și  $P_2$ .

Mulțimea punctelor dreptei care trece prin punctele  $P_1$  și  $P_2$  este:

$$\{(tx_1 + (1-t)x_2, ty_1 + (1-t)y_2) \mid t \text{ real}\}$$

unde diferite valori ale lui  $t$  furnizează coordonatele unor puncte diferite de pe dreaptă. Pentru a determina coordonata  $y$  a intersecției determinăm mai întâi din expresia lui  $x$  expresia lui  $t$ :

$$x = tx_1 + (1-t)x_2$$

$$t = (x - x_2) / (x_1 - x_2),$$

deci punctul de intersecție va fi  $(x, ty_1 + (1-t)y_2)$

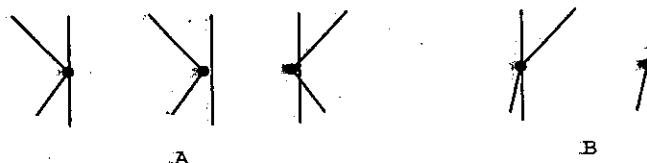
Se ivescă o mică problemă și anume faptul că în formulele de mai sus poate să apară împărțirea la 0 în cazul în care  $x_1=x_2$ . Dar dacă presupunem faptul că vom determina valorile lui  $t$  doar pentru  $x$  între  $x_1$  și  $x_2$ , acest lucru nu se poate întâmpla. Utilizând aceste considerații, algoritmul devine:

```

int i, intersectii=0;
for (i=0; i<n; i++)
    if (P[i].x < M.x && M.x < P[i+1].x || P[i].x > M.x && M.x > P[i+1].x)
        {  $t = (M.x - P[i+1].x) / (P[i].x - P[i+1].x);$ 
          $cy = t * P[i].y + (1-t) * P[i+1].y;$ 
         if (M.y == cy) return (pe_latura)
             else if (M.y > cy) intersectii++;
        }
    if (intersectii%2) return (interior);
    else return (exterior);

```

Să vedem acum ce se întâmplă în cazul în care semidreapta trece exact prin unul dintre vârfurile poligonului. În cazul în care semidreapta este doar „tangentă” la poligon în acel vârf (cazul A din figură), această intersecție nu se numără (nu este luată în considerare), în timp ce, în cazul în care semidreapta „traversează” vârful, intersecția va fi numărată.



Pentru a rezolva aceste situații, ne bazăm pe următoarea observație: dacă am mutat punctul  $M$  puțin mai la dreapta, acest lucru nu ar influența apartenența lui la interiorul sau exteriorul poligonului, în schimb semidreapta nu ar mai intersecta vârful poligonului sau intersecția se va face în două puncte (A). Dacă însă semidreapta intersectează vârful în modul indicat în (B), atunci prin deplasarea la dreapta a lui  $M$ , semidreapta va intersecta una dintre laturile care conțin acest vârf.

De fapt, în program nu vom face mutarea punctului  $M$ , ci vom testa coordonatele  $x$  ale celor două puncte vecine  $P[i-1].x$ , respectiv  $P[i+1].x$ , și, în funcție de relația existentă, vom incrementa sau nu numărul de intersecții.

```
#include <stdio.h>
#define exterior -1
#define pe_latura 0
#define interior 1
#define NMAX 200

typedef struct { double x, y; } Punct;
typedef Punct Poligon[NMAX];
Poligon P;
Punct M;
int n;

int int_pe_ext(void)
{ int i, intersectii=0;
  double t, cy;
  P[n]=P[0];
  for (i=0; i<n; i++)
    { if ((P[i].x<M.x && M.x<P[i+1].x) ||
        (P[i].x>M.x && M.x>P[i+1].x))
        { t=(M.x-P[i+1].x)/(P[i].x-P[i+1].x);
          cy=t*P[i].y+(1-t)*P[i+1].y;
          if (M.y==cy) return pe_latura; //orizontala
          else
            if (M.y>cy) intersectii++;
        }
    if (P[i].x==M.x && P[i].y>=M.y && P[i+1].y<=M.y)
      return (pe_latura); //pe verticala, P[i] sus
    if (P[i].x==M.x && P[i].y<=M.y)
      { if (P[i].y==M.y) return pe_latura;
        if (P[i+1].x==M.x)
          { if ((P[i].y<=M.y && M.y<=P[i+1].y) ||
              P[i].y>=M.y && M.y>=P[i+1].y)
            return pe_latura; //verticala, P[i] jos
        }
      }
  }
  return intersectii;
}

int main()
{ int i, v;
  freopen("puncte.in", "r", stdin);
  scanf("%d", &n);
  for (i=0; i<n; i++) scanf("%lf %lf", &P[i].x, &P[i].y);
  scanf("%lf %lf", &M.x, &M.y);
  v=int_pe_ext();
  printf("%s\n", v>0?"interior":v<0?"exterior":"pe latura");
  return 0;
}
```

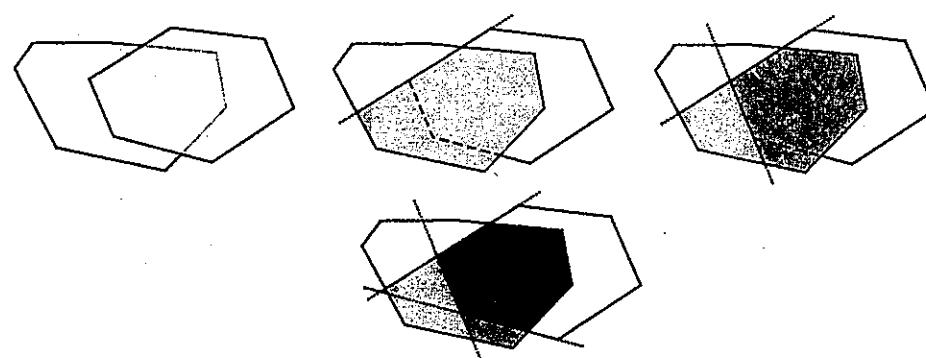
```
else //testare caz (B)
  if (P[i+1].x>M.x) intersectii++;
  if (P[i-1].x>M.x) intersectii++;
}
if (intersectii%2) return interior;
return exterior;
}

int main()
{ int i, v;
  freopen("puncte.in", "r", stdin);
  scanf("%d", &n);
  for (i=0; i<n; i++) scanf("%lf %lf", &P[i].x, &P[i].y);
  scanf("%lf %lf", &M.x, &M.y);
  v=int_pe_ext();
  printf("%s\n", v>0?"interior":v<0?"exterior":"pe latura");
  return 0;
}
```

### Intersecția a două poligoane convexe

Având două poligoane convexe date prin coordonatele vârfurilor lor în ordine trigonometrică, să se determine intersecția celor două poligoane. Intersecția va fi dată prin mulțimea vârfurilor aparținând celor două poligoane care determină poligonul intersecție.

Problema pare la prima vedere destul de complicată. O soluție posibilă ar fi să pornim pe rând cu unul dintre poligoane ca poligon fix și să îl intersectăm cu semiplanul determinat de dreapta-suport a fiecareia dintre laturile celuilalt poligon, reținând la fiecare intersecție efectuată punctele de intersecție nou apărute și eliminând vârfurile care nu mai sunt necesare. În următoarea imagine sunt ilustrați primii pași ai algoritmului:

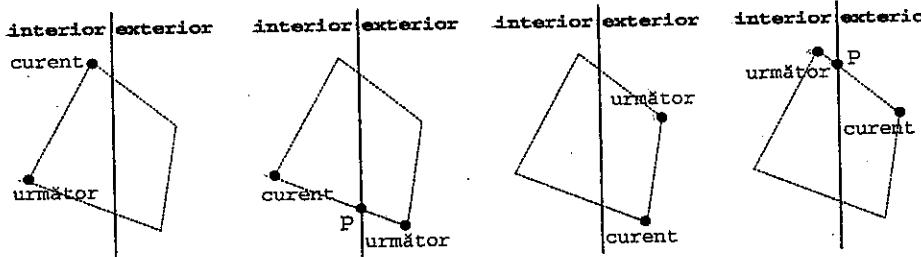


Considerăm că vârfurile poligoanelor sunt reținute în liste simplu înălțuite.

Se caută un punct interior ambelor poligoane – el va apărea sigur intersecției și față de el se va calcula valoarea interior/exterior. Primul punct se adaugă soluției curente numai dacă este în interior.

Parcurgem laturile celui de-al doilea poligon în ordine. Pentru fiecare latură luată în considerare se verifică toate laturile primului poligon. Vom avea doi pointeri *current* și *următor*, care vor conține două vârfuri consecutive ale poligonului care este soluția curentă. Există patru cazuri care trebuie luate în considerare:

1. Dacă ambele puncte sunt în interiorul semiplanului, atunci soluție curentă nu se adăugă punctul *următor*.
2. Dacă punctul *current* este în interior, iar punctul *următor* este în exterior, atunci soluție curentă nu se adăugă punctul de intersecție P.
3. Dacă ambele puncte sunt în exterior, procesăm următoarea pereche.
4. Dacă punctul *current* este în exterior și punctul *următor* este în interior, atunci soluție curentă nu se adăugă punctul de intersecție P și punctul *următor*.



Se obține astfel un algoritm de complexitate  $O(n^2)$ .

## 6.4. Geometria dreptunghiului

### Intersecția a două dreptunghiuri

Fiind date două dreptunghiuri prin coordonatele colțurilor stânga-sus și dreapta-jos, să se determine aria și perimetru intersecției celor două dreptunghiuri.

Să verificăm în primul rând dacă două dreptunghiuri se intersecțează.

Vom reprezenta un dreptunghi ca o structură astfel:

```
typedef struct {Punct SS, DJ;} DREPTUNGHII;
```

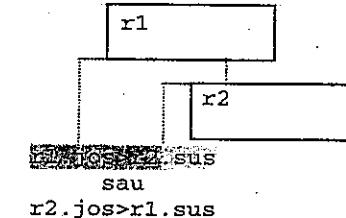
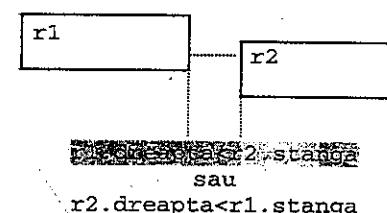
Prima idee pe care o avem este aceea de a scrie o funcție ce testează toate cazurile posibile de intersecție. Când unul dintre cele două dreptunghiuri are cel puțin un vârf în suprafața celuilalt, acestea se intersecțează. Există deci opt cazuri posibile (câte patru pentru fiecare dreptunghi). Testarea dacă un punct este interior unei suprafețe dreptunghulare se poate face cu funcția următoare:

```
int PunctInterior(Punct A, DREPTUNGHII r)
{ return (r.SS.x<=A.x && A.x<=r.DJ.x &&
         r.DJ.y<=A.y && A.y<=r.SS.y); }
```

Funcția bazată pe acestă primă abordare este:

```
int Intersect(DREPTUNGHII r1, DREPTUNGHII r2)
{ return PunctInterior(A1,r2) || PunctInterior(A2,r2) ||
       PunctInterior(A3,r2) || PunctInterior(A4,r2) ||
       // colț al lui r1 în r2?
       PunctInterior(B1,r1) || PunctInterior(B2,r1) ||
       PunctInterior(B3,r1) || PunctInterior(B4,r1);
       // colț al lui r2 în r1?
}
```

Observăm că pentru a putea implementa funcția *Intersect(DREPTUNGHII r1, DREPTUNGHII r2)* a fost necesar să „construim” din datele citite cele 8 puncte ale celor două dreptunghiuri. Ce se întâmplă însă dacă ilustrăm cazurile în care dreptunghiurile nu se intersecțează?



Se observă imediat faptul că între cele două dreptunghiuri nu există puncte de intersecție dacă extremitatea din stânga a unui dreptunghi este mai mare decât extremitatea din dreapta a celuilalt, ceea ce din dreapta este mai mică decât cea din stânga a celuilalt, ceea ce de sus mai mică decât cea de jos a celuilalt și ceea ce de jos este mai mare decât extremitatea de sus a celuilalt. Există deci 4 cazuri posibile, implementate de următoarea funcție:

```
int NuSeIntersect(DREPTUNGHII r1, DREPTUNGHII r2)
{return (r1.DJ.x<r2.SS.x || r1.SS.x>r2.DJ.x ||
         r1.DJ.y>r2.SS.y || r1.SS.y<r2.DJ.y); }
```

Cum negarea unei propoziții este echivalentă cu propoziția însăși și aplicând formula lui De Morgan  $!(a \mid\mid b) == !a \&\& !b$  obținem următoarea funcție finală, care folosește doar 4 comparații:

```
int Intersect(DREPTUNGHII r1, DREPTUNGHII r2)
{return r1.DJ.x>=r2.SS.x&&r1.SS.x<=r2.DJ.x &&
         r1.DJ.y<=r2.SS.y&&r1.SS.y>=r2.DJ.y; }
```

În acest moment ne putem ocupa de problema inițială și anume de a calcula aria și perimetru intersecției celor două dreptunghiuri. Este evident faptul că intersecția celor două dreptunghiuri este tot un dreptunghi, deci este suficient să determinăm două colțuri opuse ale acestuia.

Observăm faptul că aceste puncte se calculează simplu determinând minimumuri și maximumuri de coordonate. Astfel coordonata x a punctului stânga-sus a intersecției este maximumul dintre valorile coordonatelor x ale punctelor stânga-sus a celor două dreptunghiuri. Coordonata x a punctului dreapta-jos al intersecției este

minimumul dintre valorile coordonatelor  $x$  ale punctelor dreapta-jos ale celor două dreptunghiuri. În mod similar se calculează coordonatele  $y$  ale celor două puncte.

```
Punct StangaSus (DREPTUNGHIS r1, DREPTUNGHIS r2)
{ Punct A;
  A.x=max(r1.SS.x, r2.SS.x); A.y=min(r1.SS.y, r2.SS.y);
  return A;
}

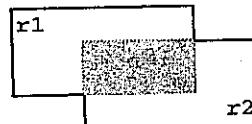
Punct DreaptaJos (DREPTUNGHIS r1, DREPTUNGHIS r2)
{ Punct A;
  A.x=min(r1.DJ.x, r2.DJ.x); A.y=max(r1.DJ.y, r2.DJ.y);
  return A;
}

double Aria (Punct A, Punct B)
{ return (B.x-A.x)*(A.y-B.y); }

double Perim (Punct A, Punct B)
{ return 2*((B.x-A.x)+(A.y-B.y)); }
```

### Reuniunea a două dreptunghiuri

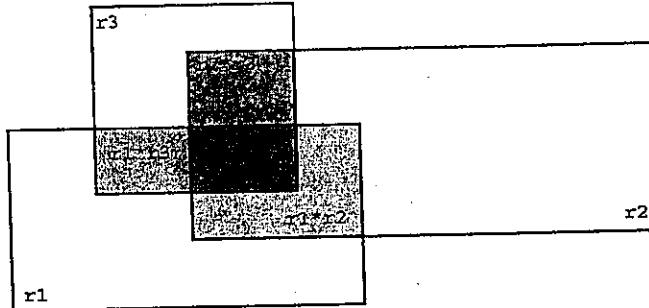
Fiind date două dreptunghiuri prin coordonatele colțurilor stânga-sus și dreapta-jos, să se determine aria și perimetru reuniiunii celor două dreptunghiuri.



Dacă vom nota cele două dreptunghiuri cu  $r_1$  și  $r_2$  ca în paragraful precedent, cu  $r_1+r_2$  reuniunea celor două dreptunghiuri, iar cu  $r_1 \cdot r_2$  intersecția lor, vom observa că se pot deduce foarte ușor formulele:

$$\begin{aligned} \text{Aria}(r_1+r_2) &= \text{Aria}(r_1) + \text{Aria}(r_2) - \text{Aria}(r_1 \cdot r_2) \\ \text{Perim}(r_1+r_2) &= \text{Perim}(r_1) + \text{Perim}(r_2) - \text{Perim}(r_1 \cdot r_2). \end{aligned}$$

Cazul în care avem trei dreptunghiuri  $r_1, r_2, r_3$  se rezolvă similar. Cea mai simplă metodă este de a „vizualiza” formula făcând un desen.



Se vor obține formulele:

$$\begin{aligned} \text{Aria}(r_1+r_2+r_3) &= \text{Aria}(r_1) + \text{Aria}(r_2) + \text{Aria}(r_3) \\ &\quad - \text{Aria}(r_1 \cdot r_2) - \text{Aria}(r_2 \cdot r_3) - \text{Aria}(r_1 \cdot r_3) \\ &\quad + \text{Aria}(r_1 \cdot r_2 \cdot r_3) \end{aligned}$$

$$\begin{aligned} \text{Perim}(r_1+r_2+r_3) &= \text{Perim}(r_1) + \text{Perim}(r_2) + \text{Perim}(r_3) \\ &\quad - \text{Perim}(r_1 \cdot r_2) - \text{Perim}(r_2 \cdot r_3) - \text{Perim}(r_1 \cdot r_3) \\ &\quad + \text{Perim}(r_1 \cdot r_2 \cdot r_3) \end{aligned}$$

Formulele de mai sus pot fi determinate și pe cale algebraică și reprezintă un caz particular al principiului incluzerii și excluderii.

### 6.5. Aplicații

#### Intersecție de segmente

Să considerăm o mulțime  $S$  formată din  $n$  segmente în plan. Să se verifice dacă există două segmente în mulțimea  $S$  care se intersecțează.

#### Soluție

O primă idee este de a considera fiecare pereche de segmente și de a verifica intersecția (complexitate timp –  $O(n^2)$ ).

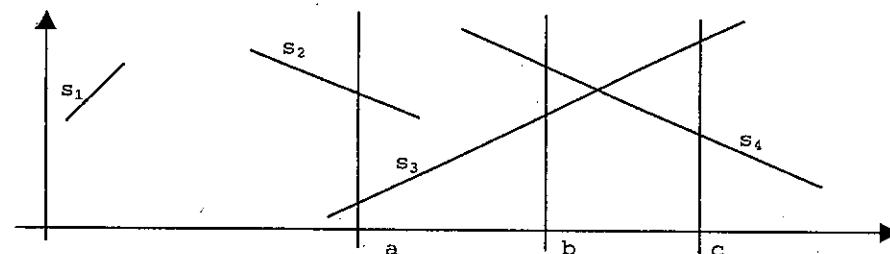
Un algoritm de complexitate  $O(n \log n)$  se bazează pe o tehnică denumită *sweeping* (baleiere, măturare). Această tehnică este interesantă deoarece poate fi aplicată pentru rezolvarea a numeroase probleme din geometria computațională.

Un algoritm de baleiere utilizează o linie verticală (paralelă cu  $Oy$ ) imaginată, denumită *sweep line* (linie de baleiere), care parcurge mulțimea dată de obiecte geometrice, de obicei de la stânga la dreapta. Pentru a parcurge obiectele este necesară o ordonare a acestora. În cazul nostru va fi necesar să ordonăm segmentele după un anumit criteriu.

Fie  $x$  o abscisă oarecare. Spunem că segmentele  $s_1$  și  $s_2$  sunt comparabile în punctul  $x$  dacă dreapta verticală care trece prin  $x$  intersecțează cele două segmente. Vom presupune pentru început că nu există segmente verticale, prin urmare, orice segment intersecțează o dreaptă verticală în cel mult un punct.

Spunem că segmentul  $s_1$  precedă segmentul  $s_2$  în punctul  $x$  ( $s_1 < s_2$ ) sau că  $s_1$  se află sub  $s_2$ , dacă ordonata punctului de intersecție al segmentului  $s_1$  cu dreapta verticală care trece prin  $x$  este  $\leq$  decât ordonata punctului de intersecție al segmentului  $s_2$  cu verticala prin  $x$ .

#### Exemplu



În figura precedentă segmentul  $s_1$  nu este comparabil cu nici unul dintre celelalte segmente. Segmentele  $s_2$  și  $s_3$  sunt comparabile (de exemplu,  $s_3 <_s s_2$ ). De asemenea, segmentele  $s_3$  și  $s_4$  sunt comparabile ( $s_3 <_s s_4$ , dar  $s_4 <_s s_3$ ).

*Observațiile cheie pe care se bazează algoritmul nostru sunt :*

1. Două segmente  $s_1$  și  $s_2$  se intersectează dacă și numai dacă există două abscise a și b, astfel încât  $s_1 <_s s_2$  și  $s_2 <_s s_1$ .
2. În ordonarea segmentelor, două segmente care se intersectează vor fi consecutive.

Un algoritm de baleiere va utiliza întotdeauna două structuri de date :

1. *Sweep\_line\_status* (starea curentă a liniei de baleiere) – reține ordinea obiectelor geometrice intersectate de *sweep line*, în poziția curentă. De la o poziție la alta, mulțimea obiectelor geometrice comparabile reținută în *sweep\_line\_status* se schimbă, prin urmare, această structură de date trebuie să suporte următoarele operații :

- *Insert(s, SLS)* – inserarea unui segment în *sweep\_line\_status*;
- *Delete(s, SLS)* – eliminarea unui segment din *sweep\_line\_status*.

Implementarea eficientă a acestor operații se realizează cu o structură de date de tip dicționar.

2. *Event\_point\_schedule* – o secvență de abscise (în ordinea de la stânga la dreapta), reprezentând pozițiile „de oprire” ale liniei de baleiere (pozițiile în care se va analiza starea curentă a liniei de baleiere). Pentru anumite probleme, această structură de date se determină în mod dinamic, în timpul execuției algoritmului. Această structură de date trebuie să suporte următoarele operații :

- *Min(E)* – extrage cel mai mic element din E;
- *Insert(x, E)* – inserează în E abscisa x;
- *Member(x, E)* – verifică dacă abscisa x aparține mulțimii E.

Implementarea eficientă a acestor operații se realizează cu o structură de date de tip coadă cu prioritate.

În cazul nostru, *sweep\_line\_status* (SLS) va reține segmente, în ordinea dată de intersecția lor cu verticala din poziția curentă. Pentru a detecta eventualele intersecții sunt necesare operații :

1. *ABOVE(s, SLS)* – determină segmentul situat imediat deasupra segmentului s în *sweep\_line\_status*;
2. *BELLOW(s, SLS)* – determină segmentul situat imediat deasupra segmentului s în *sweep\_line\_status*.

*Event\_point\_schedule* (E) este mulțimea formată din extremitățile celor n segmente, în ordinea absciselor. În cazul în care există segmente verticale (deci cele două extremități ale segmentului au aceeași abscisă) se consideră mai întâi două extremități de început (cu ordonata mai mică), apoi cele de sfârșit.

#### Algoritm în pseudocod

1. Se ordonează extremitățile segmentelor după abscisă (în cazul în care există segmente verticale, adică cele două extremități au aceeași abscisă, se plasează mai întâi extremitatea de început, cea cu ordonata mai mică, apoi cea de sfârșit).

2. Se parcurg în ordine extremitățile segmentelor. Pentru fiecare extremitate există două cazuri.

Dacă p este extremitate inițială a segmentului s, atunci :

- se inserează s în linia de stare (*Insert(s, SLS)*);
- se verifică dacă există un segment deasupra lui s (*Above(s, SLS)*) sau dedesubtul lui s (*Below(s, SLS)*) și acesta se intersectează cu s (în caz afirmativ am detectat o intersecție și algoritmul se oprește).

Dacă p este extremitate finală a segmentului s atunci :

- se verifică dacă segmentul situat imediat sub s și segmentul de deasupra lui s în linia de stare există și se intersectează (în caz afirmativ am detectat o intersecție, algoritmul se oprește);
- elimin s din linia de stare (*Delete(s, SLS)*).

Timpul de execuție al acestui algoritm este  $O(n \log n)$  dacă sortarea extremităților segmentelor se realizează optim ( $O(n \log n)$ ) și dacă implementarea SLS permite executarea operațiilor *Insert*, *Delete*, *Above*, *Below* în timp logaritmic.

#### Observație

Tehnica *sweeping* poate fi utilizată cu succes pentru a rezolva această problemă și pentru cazul detectării unei intersecții pentru o mulțime de discuri sau de dreptunghiuri cu laturile paralele cu axele. De asemenea, algoritmul poate fi adaptat pentru a determina aria/permîetrul reuniunii/intersecției a n dreptunghiuri cu laturile paralele cu axele.

#### Robot

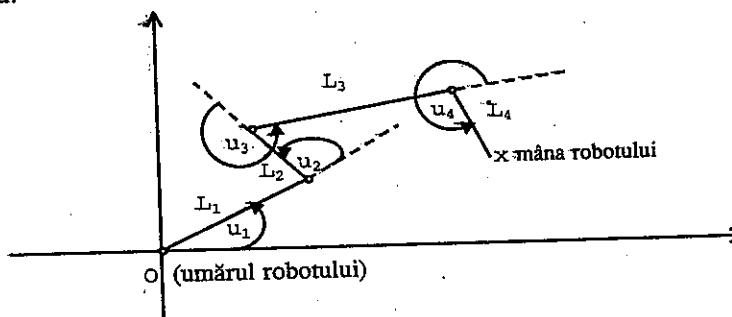
Lucrează la o firmă care produce microprocesoare. Pentru asamblarea microprocesoarelor, firma utilizează un robot constituit dintr-un singur braț. Brațul este fixat la unul dintre capete („umărul”), într-un punct plasat în centrul platformei de lucru, iar la celălalt capăt are un dispozitiv de lungime neglijabilă cu care poate „culege” componentele de pe platforma de lucru („mâna”). Brațul se poate mișca numai în plan orizontal, deasupra platformei de lucru.

Brațul este constituit dintr-o succesiune de N segmente rigide de lungimi  $L_1, L_2, \dots, L_N$ , conectate prin puncte de articulație. Mai exact, segmentul 1 este conectat printr-un punct de articulație în umărul robotului, segmentul 2 este conectat printr-un punct de articulație cu segmentul 1, ..., segmentul N este conectat printr-un punct de articulație cu segmentul N-1 și are la celălalt capăt „mâna”. Un punct de articulație permite rotația liberă (la orice unghi) a segmentului conectat în acel punct de articulație.

Pentru asamblarea unui microprocesor, robotul trebuie să culeagă succesiv componentele acestuia de pe platforma de lucru. Fiecare componentă are o poziție bine determinată pe platforma de lucru, prin coordonatele sale relativ la un sistem de coordonate cartezian, cu centru în umărul robotului.

Rolul dumneavoastră în firmă este de a programa mișările robotului. În acest scop, pentru fiecare componentă pe care robotul o va culege trebuie să specificați „configurația” brațului robotului care să permită atingerea componentei respective (mâna robotului să fie plasată deasupra poziției în care se află componenta).

Configurația brațului robotului este definită de unghiurile dintre segmentele brațului rigid.



#### Cerință

Scrieți un program care, pentru o poziție dată, determină o configurație pentru brațul robotului care să-i permită acestuia să culeagă componenta din poziția respectivă, dacă este posibil.

#### Date de intrare

Fișierul de intrare `robot.in` conține pe prima linie un număr natural  $N$  care reprezintă numărul de segmente din care este format brațul robotului. Pe fiecare dintre următoarele  $N$  linii se află câte un număr natural. Numărul aflat pe linia  $i+1$  este lungimea celui de-al  $i$ -lea segment al brațului robotului. Pe ultima linie se află două numere întregi  $x$  și  $y$ , separate prin câte un spațiu, reprezentând coordonatele poziției la care trebuie să ajungă „mâna” robotului.

#### Date de ieșire

Fișierul de ieșire `robot.out` conține o singură linie pe care se află valoarea 0 dacă nu este posibil ca mâna robotului să ajungă în poziția  $x, y$ . Dacă problema are soluție, fișierul de ieșire conține  $N$  linii. Pe linia  $i$  se află valoarea reală  $u_i$  care reprezintă unghiul dintre segmentul  $i$  și segmentul  $i-1$  (pentru orice  $i$  de la 2 la  $N$ ), iar valoarea  $u_1$  reprezintă unghiul pe care segmentul 1 îl formează în umărul robotului cu axa  $Ox$ .

#### Restriții și precizări

$$2 \leq N \leq 10000$$

$$1 \leq L_i \leq 200$$

$$0 \leq u_i < 360$$

$$-100000 \leq x, y \leq 100000$$

Unghiurile se măsoară în sens trigonometric și sunt exprimate în grade.

#### Exemple

<code>robot.in</code>	<code>robot.out</code>
3	125.6725
10	0
5	252.5424
25	
15 20	

<code>robot.out</code>
125.6725

<code>robot.in</code>
3
10
5
25
2 4

(Baraj selecție lot național, 2003)

#### Soluție

##### Accesibilitate

Regiunea accesibilă unui braț format dintr-un singur segment de lungime  $L_1$  este un cerc cu raza  $L_1$  și centru în origine (umărul robotului).

Regiunea accesibilă unui braț format din două segmente de lungimi  $L_1$ , respectiv  $L_2$ , este un inel, centrat în origine, cu raza exterioară  $L_1 + L_2$  și raza interioară  $|L_1 - L_2|$ .

Aceste rezultate simple pot fi generalizate prin inducție matematică.

Regiunea de accesibilitate a unui braț format din  $n$  segmente este un inel centrat în origine, cu raza exterioară  $r_o = L_1 + L_2 + \dots + L_n$ . Raza interioară este  $r_i = 0$  (dacă lungimea celui mai lung segment  $L_{\max}$  este mai mică decât jumătate din lungimea totală a brațului său  $r_i = 2 * L_{\max} - r_o$ ).

Acest rezultat ne permite să decidem accesibilitatea în  $O(n)$ , dar nu ne oferă informații despre configurația brațului care ne permite accesul la un anumit punct.

##### Configurația

##### Braț format din două segmente

Fie  $P$  punctul care trebuie să fie atins. Intersecțăm cercul  $C_1$  de rază  $L_1$ , centrat în origine, cu cercul  $C_2$ , de rază  $L_2$ , centrat în  $P$ .

În general pot exista 0, 1, 2 sau o infinitate de puncte de intersecție (dacă cele două cercuri coincid). Fie  $R$  un punct de intersecție. Pentru a determina configurația brațului, trebuie să calculăm unghiul dintre axa  $Ox$  și  $OR$ .

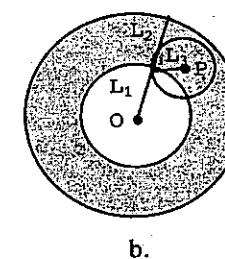
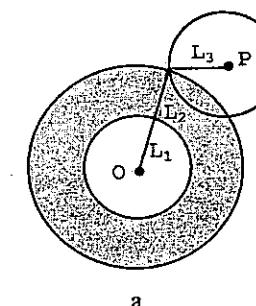
##### Braț format din trei segmente

Vom reduce cazul brațului format din trei segmente la cazul brațului format din două segmente.

Regiunea de accesibilitate a fragmentului  $(L_1, L_2)$  este un inel (să-l notăm  $A_{12}$ ).

Să examinăm intersecția dintre cercul  $C$  de rază  $L_3$ , centrat în  $P$  și inelul  $A_{12}$ . Identificăm următoarele situații:

1.  $C$  intersectează frontiera inelului  $A_{12}$ . Fie  $R$  un punct de intersecție. În acest caz, problema se poate reduce la cazul brațului format din două segmente, aliniind în același sens (figura a) sau aliniind în sensuri contrare (figura b)  $L_1$  și  $L_2$ .



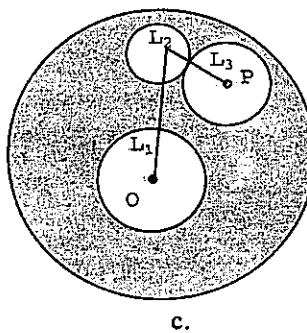
a.

b.

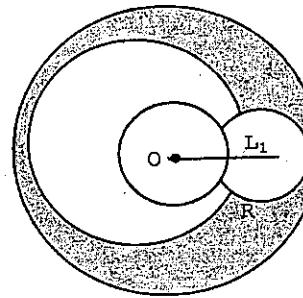
2. C nu intersectează frontieră inelului  $A_{12}$ . Identificăm în continuare următoarele două subcazuri:

C nu include originea O (figura c). Fie  $C_2$  un cerc de rază  $L_2$  tangent la C. Atunci  $L_2$  și  $L_3$  pot fi aliniate și reducem problema la cazul brațului format din două segmente.

C include originea O (figura d). În acest caz, nu există o soluție cu două segmente aliniate, dar există o soluție pentru orice valoare a unghiului  $u_1$  (de exemplu  $u_1=0$ ). Considerăm cercul  $C_1$  centrat în extremitatea finală a primului segment  $P_1$ .  $C_1$  intersectează C în punctul R. Acest punct ne dă soluția.



c.



d.

În concluzie, orice configurație cu trei segmente se poate reduce la una dintre următoarele situații cu două segmente:

- $L_1+L_2$ ,  $L_3$
- $L_1$ ,  $L_2+L_3$
- $u_1=0$ ,  $L_2$ ,  $L_3$ .

#### Braț format din n segmente

Dacă un braț format din n segmente poate atinge un punct, atunci există o configurație cu cel mult două legături „flexate” (adică numai cu unghiuri diferite de 0). Cele două legături pot fi alese la capetele segmentului median.

Segmentul median este segmentul  $L_m$  astfel încât  $L_1+L_2+\dots+L_{m-1} \leq$  jumătate din lungimea totală a segmentelor, dar  $L_1+L_2+\dots+L_m >$  decât jumătate din lungimea totală a segmentelor.

#### Demonstrație

Pentru a demonstra acest rezultat este suficient să modificați brațul „înghețând” toate legăturile, mai puțin pe cele două indicate. Brațul astfel modificat are aceeași regiune de accesibilitate.

#### Algoritm

Rezultatul precedent ne conduce la un algoritm de complexitate  $O(n)$ , în care singura parte care depinde de n este calcularea sumei lungimilor segmentelor. Celelalte operații se execută în timp constant.

```
#include <stdio.h>
#include <math.h>
#define NMax 10001
#define InFile "robot.in"
#define OutFile "robot.out"
#define sqr(x) ((x)*(x))
#define eps 0.00001

typedef struct {double x, y;} PointD;
PointD p, np;
int n, med, L[NMax];
long ro, ri;
FILE *fout;

void Citire(void);
void Raza(void);
void Rezolva_N(void);
void Rezolva_3(long, long, long);
int Rezolva_2(long, long);
double unghi0(PointD);
double unghi(PointD, PointD);
int Cercuri0(long, PointD, long);

int main ()
{double u1, u2;
Citire ();
fout=fopen(OutFile, "wt");
Raza();
if (sqr(p.x)+sqr(p.y)>sqr(ro) || sqr(p.x)+sqr(p.y)<sqr(ri))
{fprintf(fout,"0\n");
fclose(fout); return 0; }
if (n==2)
{Rezolva_2(L[1],L[2]);
u1=unghi0(np);
fprintf(fout,"% .4lf\n",u1);
u2=unghi(np, p);
fprintf(fout,"% .4lf\n",u2);
fclose(fout);
return 0; }
Rezolva_N();
return 0;
}

void Citire (void)
{
int i;
FILE *fin=fopen(InFile, "rt");
fscanf(fin,"%d",&n);
for (i=1; i<=n; i++) fscanf(fin, "%d", L+i);
fscanf(fin,"%lf %lf", &p.x, &p.y);
fclose(fin);
}
```

```

void Raza()
{
    int i, max=-1;
    ro=ri=0;
    for (i=1; i<=n; i++)
        {ro+=L[i];
         if (max<L[i]) max=L[i]; }
    if (max>ro/2) ri=2*max-ro;
}

void Rezolva_N()
{
    long sum=0;
    for (med=1; med<=n; med++)
        if (sum+L[med]>ro/2) break;
        else sum+=L[med];
    Rezolva_3(sum, L[med], ro-sum-L[med]);
}

void Rezolva_3(long l1, long l2, long l3)
{
    int i;
    double u1, u2;
    PointD temp;
    if (Rezolva_2(l1+l2,l3))
    {
        u1=unghi0(np); fprintf(fout,"%4lf\n",u1);
        for (i=2; i<=med; i++) fprintf(fout,"0\n");
        u2=unghi(np,p); fprintf(fout,"%4lf\n",u2);
        for (i=med+2; i<=n; i++) fprintf(fout,"0\n");
        fclose(fout);
        return; }
    if (Rezolva_2(l1, l2+l3))
    {
        u1=unghi0(np);
        fprintf(fout,"%4lf\n",u1);
        for (i=2; i<med; i++) fprintf(fout,"0\n");
        u2=unghi(np, p); fprintf(fout,"%4lf\n",u2);
        for (i=med+1; i<=n; i++) fprintf(fout,"0\n");
        fclose(fout);
        return; }
    for (i=1; i<med; i++) fprintf(fout,"0\n");
    p.x-=l1;
    Rezolva_2(l2,l3);
    u1=unghi0(np); fprintf(fout,"%4lf\n",u1);
    u2=unghi(np, p); fprintf(fout,"%4lf\n",u2);
    p.x+=l1; np.x+=l1;
    for (i=med+2; i<=n; i++) fprintf(fout,"0\n");
    fclose(fout);
    return;
}

int Rezolva_2(long l1, long l2)
{
    int nsol;
    nsol=Cercuri0(l1, p, l2);
    return nsol;
}

```

```

int Cercuri0(long r1, PointD c2, long r2)
{
    /*determina punctul de intersectie a doua cercuri
    primul cerc este centrat in origine si are raza r1;
    al doilea cerc este centrat in punctul c2 si are raza r2;
    functia returneaza numarul de puncte de intersectie */
    {
        long dc1c2=sqr(c2.x)+sqr(c2.y); //distanta dintre centre
        long rplus=sqr(r1+r2), rminus=sqr(r1-r2);
        double f, a2, sint, cost;
        PointD q;
        if (dc1c2>rplus || dc1c2<rminus) //cercuri nesecante
            return 0;
        if (dc1c2==rplus) //cercuri tangente exterior
            {f=r1/(double)(r1+r2);
             np.x=f*c2.x;
             np.y=f*c2.y;
             return 1; }
        if (dc1c2==rminus) //cercuri tangente interior
            {if (!rminus) //cercurile coincid
                {np.x=r1;
                 np.y=0;
                 return 3; }
            f=r1/(double)(r1-r2);
            np.x=f*c2.x;
            np.y=f*c2.y;
            return 1; }
        //doua puncte de intersectie; rotim c2 a.i. sa fie pe Ox
        a2=sqrt(dc1c2); cost=c2.x/a2; sint=c2.y/a2;
        q.x=(a2+(sqr(r1)-sqr(r2))/a2)/2;
        q.y=sqrt(sqr(r1)-sqr(q.x));
        np.x=cost*q.x-sint*q.y;
        np.y=sint*q.x+cost*q.y;
        return 2;
    }

    double unghi0(PointD np)
    {
        if (np.y>=0)
            return 180*atan2(np.y,np.x)/M_PI;
        return 360-180*atan2(-np.y,np.x)/M_PI;
    }

    double unghi(PointD np, PointD p)
    {
        PointD p1;
        double d;
        p1.x=p.x-np.x;
        p1.y=p.y-np.y;
        d=unghi0(p1)-unghi0(np);
        if (d<0) d+=360;
        return d;
    }
}

```

## 6.6. Probleme propuse

### 1. S.O.S. AEDARO98

Începând cu anul de grație 8991, a fost accelerat procesul de colonizare a planetelor care îndeplineau condiții de terraformare. Planeta AEDARO98 a fost populată de Klingonieni și Borgi. Din păcate, în scurt timp au apărut conflicte între ei. Autoritatea interstelară a decis să traseze o frontieră, care să separe coloniile Klingonienilor de coloniile Borgilor.

Cunoscând că există  $n_1$  coloane Klingoniene, respectiv  $n_2$  coloane Borgiene, specificate prin coordonatele lor carteziene referitoare la un sistem de coordinate ortogonal cu centru în mijlocul hărții planare, verificați dacă este posibilă separarea Klingonienilor de Borgi printr-o frontieră rectilinie.

#### Date de intrare

Fișierul de intrare aedaro.in conține pe prima linie  $n_1$ , reprezentând numărul de coloane Klingoniene ( $1 \leq n_1 \leq 100$ ). Pe următoarele  $n_1$  linii sunt specificate coordonatele carteziene ale coloanelor Klingoniene. Pe linia  $n_1+2$  se află numărul natural  $n_2$ , reprezentând numărul de coloane Borgiene ( $1 \leq n_2 \leq 100$ ). Pe următoarele  $n_1$  linii sunt specificate coordonatele carteziene ale coloanelor Borgiene.

#### Date de ieșire

Pe prima linie a fișierului de ieșire aedaro.out se va scrie cuvântul POSIBIL, respectiv IMPOSSIBIL, după caz. În cazul în care coloanele pot fi separate printr-o frontieră rectilinie, pe cea de-a doua linie veți scrie 3 valori reale, cu trei zecimale, separate printr-un spațiu a b c, cu semnificația „dreapta a·x+b·y=c separă coloanele Klingoniene de coloanele Borgiene”.

#### Exemplu

aedaro.in	aedaro.out	aedaro.in	aedaro.out
3	POSIBIL	3	IMPOSSIBIL
0 5	1.000 0.000 6.000	0 5	
0 0		0 0	
5 0		5 0	
2		2	
10 0		10 2	
8 0		2 2	

(Olimpiada Națională de Informatică, Oradea, 1998)

### 2. Moșia lui Păcală

Păcală a primit, aşa cum era învoiala, un petec de teren de pe moșia boierului. Terenul este împrejmuit complet cu segmente drepte de gard ce se sprijină la ambele capete de către un par zdravăn. La o nouă prinsoare, Păcală ieșe iar în câstig și capete de către un par zdravăn. La o nouă prinsoare, Păcală ieșe iar în câstig și primește dreptul să strămute niște pari, unul către unul, cum i-o fi voia, astfel încât să-și extindă suprafața de teren. Dar învoiala prevede că fiecare par poate fi mutat în orice direcție, dar nu pe o distanță mai mare decât o valoare dată (scrisă pe fiecare par) și fiecare segment de gard, fiind cam subred, poate fi rotit și prelungit de la un singur capăt, celălalt rămânând nemîșcat.

Cunoscând pozițiile inițiale ale parilor și valoarea inscrisă pe fiecare par se cere suprafață maximă cu care poate să-și extindă Păcală proprietatea. Se știe că pari sunt dări într-o ordine oarecare, pozițiile lor inițiale sunt date prin numere întregi de cel mult trei cifre, distanțele pe care fiecare par poate fi deplasat sunt numere naturale strict pozitive și că figura formată de terenul inițial este un poligon neconcav.

#### Date de intrare

Fișierul de intrare mosia.in conține pe prima linie un număr natural  $n$ , reprezentând numărul de pari. Pe următoarele  $n$  linii sunt descriși pari, câte un par pe o linie. Pentru fiecare par sunt precizate trei valori întregi  $x$   $y$   $d$ , reprezentând coordonatele sale inițiale, precum și distanța cu care poate fi mutat parul.

#### Date de ieșire

În fișierul de ieșire mosia.out se scrie un număr real cu patru zecimale ce reprezintă suprafață maximă cu care se poate mări moșia.

#### Restricții și precizări

$3 < N \leq 200$   
 $-1000 < x, y < 1000$   
 $0 < d \leq 20$

Poligonul neconcav se definește ca un poligon convex cu unele vârfuri coliniare. Pozițiile parilor sunt date într-o ordine oarecare.

Poligonul obținut după mutarea parilor poate fi concav.

Pozițiile finale ale parilor nu sunt în mod obligatoriu numere naturale.

#### Exemplu

mosia.in	mosia.out
4	30.0000
-3 0 2	
3 0 3	
0 6 2	
0 -6 6	

#### Explicație

Prin mutarea parilor 1 și 2 cu câte 2, respectiv 3 unități, se obține un teren având suprafață cu 30 de unități mai mare decât terenul inițial.



(Olimpiada Județeană de Informatică, 2004)

#### 3. Robot

Un robot punctiform se poate deplasa în plan în linie dreaptă în orice direcție. Robotul se găsește într-o poziție inițială  $S$  și trebuie să ajungă într-o poziție finală  $F$ , evitând coliziunile cu obstacolele existente în teren. Obstacolele sunt suprafețe poligonale convexe, cu interioarele și frontierele disjuncte. Spunem că robotul a intrat în coliziune cu un obstacol dacă poziția sa devine interioară obstacolului. Prin urmare, dacă robotul se deplasează de-a lungul unui obstacol, nu intră în coliziune cu acesta.

Scriți un program care să determine cel mai scurt traseu pe care robotul îl poate urma de la poziția sa inițială  $S$  la poziția sa finală  $F$ , fără a intra în coliziune cu nici un obstacol.

Traseul va fi precizat prin succesiunea punctelor critice (punctul inițial, punctele în care robotul își schimbă direcția și punctul final). Lungimea traseului este egală cu suma lungimilor segmentelor care îl compun.

*Date de intrare*

Fișierul de intrare `robot.in` conține pe prima linie `xs ys`, coordonatele poziției initiale a robotului. Pe cea de-a doua linie se află `xf yf`, coordonatele poziției finale a robotului. Pe a treia linie se află `n`, numărul de obstacole. Urmează descrierea celor `n` obstacole. Pentru fiecare obstacol este precizat pe prima linie un număr natural `m`, reprezentând numărul de vârfuri ale obstacolului. Pe următoarele `m` linii sunt scrise coordonatele celor `m` vârfuri ale obstacolului.

*Date de ieșire*

Fișierul de ieșire `robot.out` conține traseul robotului codificat ca o succesiune de puncte între care robotul se mișcă în linie dreaptă. Pe prima linie este scris `nr`, numărul de puncte de pe traseu. Pe următoarele linii sunt scrise coordonatele punctelor de pe traseu (începând cu punctul inițial și încheind cu punctul final).

*Restricții*

$$0 \leq n \leq 50$$

Obstacolele au cel mult 200 de vârfuri.

Coordonatele vârfurilor obstacolelor sunt numere reale,  $|x_i|, |y_i| \leq 100000$ .

Punctul inițial și cel final nu se află în interiorul nici unui obstacol.

Coordonatele se vor afișa în fișierul de ieșire cu trei zecimale semnificative.

*Exemplu*

<code>robot.in</code>	<code>robot.out</code>
10 5	3
-10 -10	10 5
1	10.5 0
3	-10 -10
0 0	
0 10	
10.5 0	

(Olimpiada Națională de Informatică, Bacău, 2001)

**4. Galerie**

O galerie de artă modernă funcționează într-o încăpere a cărei podea are forma unui poligon cu  $N$  vârfuri. Pentru a asigura securitatea, directorul galeriei dorește să instaleze camere video de supraveghere. Camerele video achiziționate vor fi fixate în anumite puncte din încăpere și ele se pot roti, astfel încât câmpul lor de vizibilitate poate fi de 360 grade.

Fiind un fost profesor de matematică, directorul a demonstrat că un număr  $M$  de camere video sunt suficiente pentru a supraveghea întreaga încăpere.

Cunoscând forma încăperii scrieți un program care să stabilească pozițiile în care pot fi fixate maximum  $M$  camere astfel încât întreaga încăpere să fie supravegheată.

*Date de intrare*

Fișierul de intrare `galerie.in` conține pe prima linie două numere naturale  $N M$ , reprezentând numărul de vârfuri ale poligonului și respectiv numărul maxim de camere video. Pe următoarele  $N$  linii sunt specificate în sens trigonometric vârfurile poligonului. Pentru fiecare vârf sunt scrise pe o linie abscisa și ordonata separate prin spațiu (numere întregi din intervalul  $[-30000, 30000]$ ).

*Date de ieșire*

Fișierul de ieșire `galerie.out` va conține pe prima linie un număr natural  $Nr$  reprezentând numărul de camere video instalate. Pe următoarele  $Nr$  linii vor fi specificate coordonatele carteziene ale punctelor în care se instalează camere video (în ordinea abscisă-ordonată).

*Restricții și precizări*

$$3 \leq N \leq 100$$

Coordonatele punctelor în care se instalează camere video vor fi cu trei zecimale. Camerele video nu își blochează reciproc vizibilitatea.

*Exemplu*

<code>galerie.in</code>	<code>galerie.out</code>
5 1	1
10 5	60.000 10.000
100 5	
80 120	
60 20	
30 120	

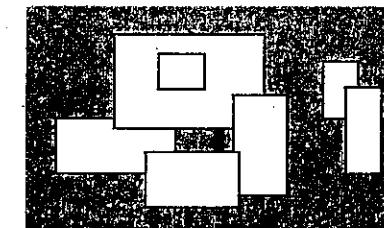
(Baraj selecție lot național, 2002)

**5. Dreptunghiuri**

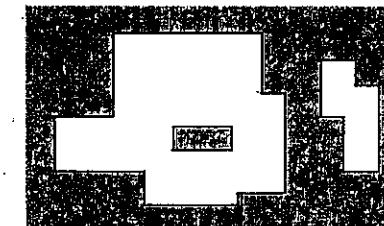
Pe un perete se lipesc mai multe postere, fotografii și alte desene de forme dreptunghiarale. Laturile sunt paralele cu axele de coordonate. Dreptunghiurile pot să se suprapună parțial sau integral. Lungimea marginilor (contururilor) reuniunii tuturor dreptunghiurilor se va numi perimetru.

Să se calculeze perimetru zonei acoperite de dreptunghiuri.

În figură avem un exemplu cu șapte dreptunghiuri :



Perimetru este suma lungimilor segmentelor desenate în figura următoare :



*Date de intrare*

Pe prima linie a fișierului de intrare drept.in se află un număr care reprezintă numărul figurilor dreptunghiulare lipite pe perete. Pe următoarele linii sunt descrise figurile dreptunghiulare, câte o figură pe o linie. Pentru fiecare dreptunghi sunt specificate coordonatele colțului stânga-jos și dreapta-sus a dreptunghiurilor. Coordonatele colțurilor sunt specificate în ordinea abscisă, ordonată.

*Date de ieșire*

Fișierul de ieșire drept.out trebuie să conțină o singură linie pe care este scris un număr întreg nenegativ reprezentând perimetru determinat.

*Restricții*

$0 \leq$  numărul de dreptunghiuri  $< 5000$

Toate coordonatele sunt întregi aparținând intervalului  $[-10000, 10000]$ .

Valoarea numerică a rezultatului necesită o reprezentare pe 32 de biți.

*Exemplu*

drept.in	drept.out
7	228
-15 0 5 10	
-5 8 20 25	
15 -4 24 14	
0 -6 16 4	
2 15 10 22	
30 10 36 20	
34 0 40 16	

*Observație*

Exemplul corespunde figurii din enunț.

(Olimpiada Internațională de Informatică, Portugalia, 1998)

*6. Ajutor*

Dacă te-ai rănit, nu folosi Sprite să tratezi rana. Cel mai bine este să fugi la cel mai apropiat post de prim ajutor. Norocul tău este că ai o hartă din care poți afla coordonatele carteziene ale posturilor de prim ajutor. Ghinionul este că, din cauza durerii, nu poți să fugi direct spre un post, ci numai pe direcțiile nord-sud și est-vest. Ca să știi dacă mai are sens să fugi sau să te bucuri de ultimele clipe de viață, cel mai bine este să evaluezi distanța până la cel mai apropiat post de prim ajutor. Cel mai apropiat post este cel pentru care distanța Manhattan este minimă. De data aceasta ai scăpat pentru că timpul necesar ajungerii până la post a fost suficient; însă îți pui întrebarea: dacă accidentul s-ar fi întâmplat în alt loc, ai fi scăpat?

*Cerință*

Se consideră N puncte în plan, reprezentând posturile de prim ajutor, și alte M puncte reprezentând posibile locații ale accidentului. Se cere pentru fiecare dintre cele M puncte distanța Manhattan până la cel mai apropiat post.

*Date de intrare*

Fișierul ajutor.in conține pe prima linie numerele întregi N și M, separate prin spațiu. Pe următoarele N linii se află câte două numere întregi, separate prin spațiu, reprezentând coordonatele fiecărui post de prim ajutor. Pe următoarele M linii se află câte două numere întregi, separate printr-un spațiu, reprezentând coordonatele punctelor de accident. Coordonatele se dau în ordinea abscisă, ordonată.

*Date de ieșire*

Fișierul ajutor.out va conține M linii, cu câte un număr pe fiecare linie, reprezentând distanța minimă până la cel mai apropiat post de prim ajutor.

*Restricții și precizări*

$0 < N < 401$

$0 < M < 500001$

Oricare coordonată este un număr întreg din intervalul  $[0, 32000]$ .

Distanța Manhattan este cel mai scurt drum între două puncte, mergând doar pe direcții paralele cu axele de coordonate, adică  $|x_1 - x_2| + |y_1 - y_2|$ , unde  $(x_1, y_1), (x_2, y_2)$  sunt coordonatele celor două puncte.

În fișierul de intrare pot exista puncte cu aceleași coordonate.

*Exemplu*

ajutor.in	ajutor.out
4 4	2
1 1	0
5 5	4
1 5	1
5 1	
0 0	
1 1	
3 3	
4 1	

(Baraj selecție lot național, 2003)

## 7. Soluții și indicații

### 1. Teoria grafurilor

#### 1. Turn

Putem asocia problemei un graf neorientat astfel: vârfurile grafului sunt paharele; dacă se execută mutarea a  $\rightarrow$  b, atunci există muchie în graf între a și b. Turnurile sunt componentele conexe ale grafului, prin urmare:

- a. descompunem graful în componente conexe;
- b. pe parcursul descompunerii determinăm  $\max_1$  și  $\max_2$  (o componentă conexă cu număr maxim de elemente și cea de a doua componentă conexă cu număr maxim de elemente);
- c. unificăm  $\max_1$  și  $\max_2$  realizând o mutare în care să apară un element din  $\max_1$  și un element din  $\max_2$  (dacă graful este conex, putem afișa mutarea 1  $\rightarrow$  1).

#### 2. Program

Putem asocia problemei un graf orientat în care vârfurile sunt instrucțiunile din program. Există arc de la vârful x la vârful y dacă de la instrucțiunea x se poate trece la executarea instrucțiunii y. Vom parcurge graful asociat problemei din vârful 1 (corespunzător primei instrucțiuni din program). Vârfurile care nu au fost vizitate în timpul parcurgerii corespund instrucțiunilor care cu siguranță nu vor fi executate.

#### 3. APM

Adăugarea unei muchii într-un arbore generează un ciclu. Presupunem că introducem o muchie între nodul i și nodul j. Pentru ca APM-ul grafului astfel obținut să nu se schimbe, trebuie ca muchia de cost maxim de pe drumul de la i la j din arborele inițial să aibă costul mai mic sau egal cu muchia introdusă. Se determină muchiile de cost maxim de pe fiecare drum dintre oricare două vârfuri ale arborelui inițial; se ordonează crescător atât vectorul acestora cât și costurile muchiilor. Fișierul de ieșire va conține toate perechile de vârfuri distincte pentru care avem o valoare mai mare sau egală cu muchia de cost maxim de pe drumul dintre cele două vârfuri.

#### 4. Zmeu

Pornind de la fiecare dintre cele 5 camere ale fetelor se determină timpul minim pentru a ajunge la Făt-Frumos, la celelalte 4 fete și la cea mai apropiată ieșire (folosind algoritmul lui Dijkstra). Generăm apoi toate permutările de ordin 5 pentru a stabili ordinea în care vor fi vizitate cele 5 fete și alegem varianța pentru care se obține minimumul.

#### 5. Graf

Se efectuează o parcurgere în lățime începând din vârful x și se determină pentru fiecare vârf i din graf  $dx[i] =$  distanță minimă de la x la i (exprimată în număr de muchii). Se determină de asemenea și un lanț de lungime minimă de la x la y. Se efectuează apoi BFS(y) și se determină pentru fiecare vârf i  $dy[i] =$  distanță minimă de la y la i. Pentru fiecare vârf aflat pe lanțul minim de la x la y verificăm dacă el se află pe orice lanț de lungime minimă de la x la y ( $dx[i]+dy[i]=dx[y]$ ).

#### 6. Trasee

Mai întâi vom efectua o parcurgere în lățime pentru a determina distanțele de la nodul x la restul nodurilor din graf. Vom păstra doar muchiile din graf care sunt între două noduri la distanțe consecutive de nodul x. Toate nodurile situate la distanță k de nodul de start vor fi unite la un nou nod y. Acum pe graful rezultat trebuie să găsim cât mai multe drumuri disjuncte din punctul de vedere al muchiilor, care pornesc din nodul x și ajung în nodul y, muchiile ce intră în nodul y pot fi folosite de mai multe ori. Pentru a rezolva problema folosim un algoritm de flux maxim, unde muchiile sunt cele din graful menționat, ele având capacitatele 1, iar muchiile ce intră în nodul y vor avea capacitatele infinit. Fluxul maxim este egal cu numărul maxim de trasee cerut în problemă.

#### 7. Grade

Pentru a garanta conexitatea se modifică algoritmul de determinare a unui graf cu secvența gradelor dată astfel:

1. Se sortează descrescător gradele.
2. Se consideră că nodul 1 are gradul  $d_1$  și este adjacente cu nodurile 2, 3, ..., dar în cazul în care există mai multe valori egale cu  $d_1$ , se consideră că 1 este adjacente cu ultimul (ca număr de ordine) dintre nodurile care au gradul  $d_1$ ; deci se va scădea 1 din  $d_2$ ,  $d_3$ , ...
3. Se repetă pasul 2 până când toate gradele devin 0.

Pentru a reduce complexitatea algoritmului, trebuie evitată parcurgerea repetată a listei de grade pentru localizarea elementelor din stânga ale palierelor; aceasta se face prin menținerea unei liste cu pozițiile de început și sfârșit ale fiecărui palier.

#### 8. Traseu

Pentru majoritatea testelor se poate aplica metoda *backtracking*. Asociem problemei un graf orientat astfel: fiecare curs va fi considerat un vârf din graf; dacă pentru studierea cursului y este necesară studierea cursului x, în graf va exista arc de la x la y.

Pentru că problema admite întotdeauna soluție pe datele de test, graful asociat nu conține circuite. Fiecare vârf din graf are asociat un cost (numărul de credite asociate cursului corespunzător vârfului). Se descompune graful dat pe niveluri. Se pornește construirea soluției selectând un vârf de pe ultimul nivel (și implicit toate vârfurile de care depinde acesta). Fie sum suma costurilor vârfurilor selectate. Se completează (generând prin *backtracking*) diferența de cost S-sum selectând vârfuri care nu au mai fost deja selectate. O optimizare importantă ar fi selectarea vârfurilor în ordinea descrescătoare a nivelurilor pe care se află (și implicit selectarea totodată a tuturor vârfurilor de care depind acestea). Analizând fișierele de intrare se pot oferi și soluții particulare dependente de structura acestor fișiere.

**9. Loc**

Se aplică algoritmul lui Dijkstra modificat astfel încât să determine și numărul de drumuri de cost minim.

**10. Trip**

Problema cere să se găsească două drumuri între două vârfuri, drumuri care să aibă cât mai puține muchii comune. Observația cheie este că într-o componentă biconexă există un ciclu între oricare două vârfuri (deci există două drumuri disjuncte). Singurele muchii care vor fi parcurse de două ori sunt muchiile critice care despart componente biconexe. Primul pas care trebuie făcut este de a determina componente biconexe ale grafului. Al doilea pas ar fi determinarea unui drum între sursă și destinație care să treacă prin cât mai puține componente biconexe. Pentru aceasta se asociază fiecărei muchii  $[a, b]$  un cost astfel:  $c[a][b]=0$  dacă  $a$  și  $b$  sunt în aceeași componentă biconexă și  $c[a][b]=1$  dacă  $a$  și  $b$  sunt în componente biconexe diferite. Pentru a determina drumul minim se poate folosi algoritmul lui Dijkstra. Pasul final constă în a elimina din graf toate componente biconexe care sunt parcurse de drumul găsit mai sus. Apoi, în subgraful obținut, se stabilește pentru fiecare muchie critică capacitatea 2, iar pentru celelalte muchii capacitatea 1. În graful obținut se caută două drumuri de ameliorare între sursă și destinație.

**11. Turism**

Problema se reduce la a afla un ciclu elementar de cost minim. Cum toate costurile sunt pozitive, se poate folosi algoritmul lui Dijkstra astfel:

- se selectază, pe rând, toate nodurile ca sursă și se aplică algoritmul lui Dijkstra;
- în arborele obținut, se adaugă pe rând toate muchiile care nu apar în arbore; se formează astfel un ciclu; fiecare ciclu format este candidat pentru ciclul elementar de cost minim.

Tot ce trebuie arătat este că ciclul astfel obținut este elementar și de cost minim. Faptul că ciclul are cost minim este evident (practic se încearcă să se închidă toate lanțurile de cost minim). Rămâne să arătăm că ciclul este elementar. Singurul caz în care ciclul care se formează nu este elementar este următorul:



Se face Dijkstra din nodul  $x$  și se adaugă o muchie între nodurile  $a$  și  $b$ . Nodul  $z$  (diferit de rădăcină) este strămoș comun al nodurilor  $a$  și  $b$ . Ciclul obținut este  $x \dots z \dots a \dots b \dots z \dots x$  (care nu este elementar). Putem fi însă siguri că nu acest ciclu va fi selectat ca ciclu final, deoarece (datorită faptului că graful are costuri pozitive), ciclul  $z \dots a \dots b \dots z$  are un cost mai mic. Acest ciclu va fi găsit cu siguranță când nodul  $z$  este ales ca sursă pentru Dijkstra.

**12. Lanț de alarmă**

Asociem problemei un graf orientat în care vâfurile sunt elevii. Dacă elevul  $x$  este succesor al elevului  $y$  atunci există arc de la  $y$  la  $x$ . Alegem un vârf de grad interior 0 și construim un drum până când ajungem într-un vârf care aparține deja drumului. Acest ultim arc va fi modificat (el va indica un vârf de grad interior 0, altul decât cel inițial).

acest mod se obține un drum, iar numărul de modificări efectuate este egal cu numărul de vârfuri cu gradul interior 0 - 1. Nu este obligatoriu ca prin acest procedeu să se obțină un lanț de alarmă. Este posibil să se obțină circuite izolate. Printr-o singură mutare, aceste circuite pot fi conectate la drumul inițial. Închiderea finală a lanțului se realizează printr-o nouă mutare. Deci în total numărul de mutări va fi egal cu numărul de vârfuri care inițial aveau gradul 0 + numărul de circuite izolate.

**13. Ziduri**

Se asociază un graf în care fiecare cameră este un vârf, iar între două camere vecine pe orizontală sau verticală există o muchie; muchia are cost 1 dacă nu există zid între cele două camere, respectiv 0 dacă există zid. Utilizând algoritmul lui Dijkstra se determină costurile minime de la camera stânga-sus la toate celelalte camere. Apoi se construiește graful orientat fără circuite al tuturor drumurilor minime de la camera stânga-sus la camera dreapta-jos. Pe acest graf se calculează cu o parcurgere drumul dintre cele două camere care folosește un număr minim de muchii formate din ziduri.

**14. Team**

Se calculează drumul minim între oricare două destinații distincte (inclusiv și punctul de plecare, nodul 1, printre acestea, chiar dacă nu este o destinație). Se calculează costul minim cerut prin metoda programării dinamice. O implementare posibilă determină:

$\text{cost}[i, j, k] = \text{costul minim pentru a transporta întregul grup omogen } \{i, \dots, j\}$  pornind din nodul care este destinația persoanei  $k$ .

$\text{cost}[i, j, k] \leftarrow \min\{\text{cost}[i, u-1, u] + \text{cost}[u+1, j, u] + x[i, u], \text{unde } u \in \{i, \dots, j\}$   
 $x[i, u] = \text{costul minim de la destinația persoanei } i \text{ la destinația persoanei } u$ .

**2. Liste înlățuitoare****Liste simplu înlățuitoare**

- $p==q->urm \quad || \quad q=p->urm$ ; 2.  $L->urm->urm->urm$ ; 3.  $L \&& L->urm \&& !L->urm->urm$ ; 5. b; 6. c; 13. a, a, b, c, c. bfec, d, b, e, c, f, c; 14. b; 15. d.

**Liste simplu înlățuitoare circulare**

- 14253; 2. Numărul de elemente din listă -1 ; 4. b.

**Liste dublu înlățuitoare**

- b; 3.  $\text{Insert}(L, \text{NULL}, 1); \quad \text{Insert}(L, L, 2); \quad \text{Insert}(L, \text{NULL}, 2); \quad \text{Insert}(L, L->urm->urm, 4);$  4.  $\text{Delete}(L, SL->pre); \quad 7. c.$

**3. Structuri de date arborescente****2. Arboare**

Vom selecta succesiv câte un vârf neterminal și vom considera acest vârf rădăcina arborelui. Un vârf este considerat potențial „rezolvabil” dacă și numai dacă are ca subarbore numai vârfuri terminale sau „fire” (subarbore care este lanț).

Cât timp nu am rezolvat toate vârfurile (există vârfuri care nu aparțin unui ciclu) și nici nu am depistat o situație nerezolvabilă execut:

1. identific vârfurile potențial rezolvabile; fie x un astfel de vârf,

2. analizez cazurile următoare

x are mai mult de 2 fiu terminali → nu există soluție

x are exact 2 fiu și aceștia sunt terminali → unesc cei doi fiu printr-o muchie; am obținut un ciclu cu 3 vârfuri care îl rezolvă pe x și cei doi fiu ai săi;

x are 2 fiu terminali dar și alte „fire”: unesc cei doi fiu terminali (rezultă un ciclu de lungime 3 în care intră x și cei doi terminali), apoi rezolv firele (unesc printr-o muchie primul și ultimul vârf de pe „fir”; evident acest lucru este posibil dacă toate firele au lungimea mai mare decât 2);

x are 1 fiu terminal și evident alte „fire”: unesc vârful terminal cu extremitatea finală a uneia dintre fire (cel de lungime 2 dacă există, oricare altul dacă nu există), apoi rezolv firele (acestea trebuie să aibă lungime mai mare decât 2)

x nu are fiu terminali, numai fire; rezolv firele (dacă există mai mult de 2 fire de lungime 2 nu există soluție), unind între ele extremitățile finale a două fire (cele de lungime 2 dacă există, sau oricare altale), iar restul le rezolv independent (unind extremitatea lor inițială cu extremitatea lor finală).

### 3. Barca

Se utilizează un *heap*.

### 4. Formă normală

Construim arborele binar asociat expresiei. Aplicăm apoi transformări în arbore pentru a aplica distributivitatea atât timp cât este posibil. Sortarea factorilor, respectiv a termenilor, o vom face la sfârșit.

### 5. asmin

Se fixează ca rădăcină nodul 1 și se calculează valorile asociate nodurilor arborelui având rădăcina în nodul 1. Pentru aceasta se va efectua o parcurgere DFS a arborelui din nodul 1. Valoarea rădăcinii  $V_1=R_1$ . Pentru fiecare alt nod, valoarea asociată lui este ceea mai mică valoare astfel încât suma valorilor până la tatăl lui în arborele cu rădăcina 1 plus această valoare să dea restul cerut la împărțirea cu numărul K. Valorile sunt atribuite nodurilor de la rădăcină către frunze.

Astfel se obține  $C_1=\sum$  valorilor nodurilor arborelui când nodul 1 este rădăcina. Pentru a calcula  $C_i$  (i diferit de 1) nu este nevoie să repetăm parcurgerea DFS pentru nodul i considerat rădăcina. În schimb, încă o parcurgere DFS din nodul 1 este necesară pentru a calcula costurile fiecărui nod. Să presupunem că vrem să calculăm  $C_i$  și am calculat deja  $C_j$ , unde j este tatăl lui i, în arborele având rădăcina fixată în nodul 1.

Atunci,  $C_i=C_j-R_j-\min(R_j, R_i)+R_i+\min(R_i, R_j)$ , unde  $\min(a, b)$  (cu  $0 \leq a, b \leq K-1$ ) calculează valoarea x minimă ( $x \geq 0$ ), astfel încât  $a+x=b$  (%K). Se alege minimumul din vectorul C și se afișează nodurile i având valoarea  $C_i$  egală cu minimumul.

### 6. Telegraf

O codificare se poate reprezenta ca un arbore binar, fiecare simbol fiind reprezentat de o frunză. Codificarea simbolului reprezintă drumul de la rădăcina la frunza corespunzătoare; muchia de la un nod la fiul său stâng este etichetată cu punct, iar muchia spre fiul drept cu linie. Se observă că simbolurile nu se pot găsi decât în frunze

(din cauza condiției de neambiguitate). Durata transmisiei este dată de suma drumurilor de la rădăcină la fiecare frunză, ponderată cu frecvența de apariție a simbolului; evident, o muchie spre fiul drept are lungimea 2, iar o muchie spre fiul stâng lungimea 1.

Vom construi arboarele optim prin programare dinamică. Vom construi (conceptual) arborele pe niveluri de sus în jos. Următoarele informații reprezintă starea:

A – câte noduri interne sunt pe nivelul L-1

B – câte noduri interne sunt pe nivelul L-2

F – câte frunze au apărut deja pe nivelele mai mici decât L.

Nu trebuie să știm efectiv nivelul L. Vom găsi optimul pentru o anumită configurație [A] [B] [F], la orice nivel s-ar afla. Bineînțeles, dacă nu știm L nu putem calcula efectiv costul unei frunze când o construim. De aceea costurile trebuie adunate „pe parcurs”: când coborâm cu un nivel în arbore, vom aduna costul unui nivel în contul tuturor frunzelor aflate mai jos decât nivelul respectiv. Evident, frunzele mai jos de nivelul L au indicii F+1, F+2, ... (dacă am ordonat simbolurile după frecvență). Deci costul unui nivel în plus va fi dat de suma frecvențelor simbolurilor începând cu F+1; aceste sume le vom calcula la începutul algoritmului. Așa cum am menționat, algoritmul construiește un astfel de tablou tridimensional prin programare dinamică. Pentru aceasta, trebuie să vedem din ce poate proveni o soluție [A] [B] [F]. Singura variabilă este numărul de frunze de pe nivelul L; pentru x frunze pe nivelul L, starea precedentă este [B-A] [A] [F-L].

### 7. Acolor

Rezolvăm problema prin programare dinamică. Arborele din problema este un arbore binar de căutare și observăm imediat că predecesorul ca ordine este cel mai din dreapta nod al subarborelui de la stânga, iar succesorul este cel mai din stânga nod al subarborelui din dreapta. Pornind de la această observație calculăm ca subprobleme numărul de moduri pentru a colora un anumit subarbore cu K culori, având fixate culoarea celui mai din stânga, culoarea rădăcinii și culoarea nodului cel mai din dreapta a acelui subarbore.

$C_{n,st,r,dr}$  = numărul de moduri pentru a colora cu K culori subarborele având ca rădăcină nodul n și colorând nodul cel mai din stânga, nodul n și nodul cel mai din dreapta cu respectiv culorile st, r, dr.

Dacă n este nod frunză,  $C_{n,st,r,dr}=1$  (pentru  $st=r=dr$ ) și (altfel),

dacă n este nod interior,  $C_{n,st,r,dr} = \sum_{i+r,j+r} C_{f1,st,i,j} * \sum_{i+r,j+r} C_{f2,i,j,dr}$  (unde f1 este fiul cel mai din stânga, iar f2 fiul cel mai din dreapta).

### 4. Pattern-matching

#### 1. Colier

Problema se rezolvă utilizând un algoritm de căutare (KMP) a unui subșir într-un sir, algoritm care va determina sirul deplasamentelor. Apar, totuși, două probleme. Fiind vorba despre un colier este posibil ca secvența de culori să înceapă la sfârșitul colierului și să continue la începutul lui. Problema se rezolvă adăugând inițial la sfârșitul colierului primele M-1 mărgele din colier. Trebuie căutate aparițiile distincte ale subșirului, ceea ce presupune eliminarea din sirul de deplasamente determinat a celorăi între care diferență este < M.

## 2. Cifru

Fiecare apariție a unui simbol din alfabet într-un sir se înlocuiește cu distanța față de precedenta apariție a aceluiași simbol (considerând sirul circular, deci pentru prima apariție a simbolului se ia distanța față de ultima apariție a aceluiași simbol). Făcând această recodificare pentru cele două siruri reducem problema la determinarea permutării circulare care duce primul sir în al doilea, ce poate fi rezolvată cu un algoritm de *pattern-matching*, dacă concatenăm primul sir cu el însuși, rezultând o complexitate  $O(n)$ . Pentru m se pot genera toate permutările mulțimii  $\{1, 2, \dots, m\}$  făcând pentru fiecare permutare o căutare (cu KMP, de exemplu), iar pentru n se poate căuta permutarea pentru  $d=0, 1, \dots, n$ .

## 5. Hashing

### 1. Broasca buclucașă

Problema cere de fapt să se determine cea mai mare mulțime de puncte coliniare și echidistante dintr-o mulțime dată de puncte. Problema se rezolvă utilizând o tabelă hash statică în care se memorează punctele; memoria utilizată este deci  $O(N)$ . Timpul pentru construcția tabelei hash este nesemnificativ; algoritmul de construcție durează în cel mai favorabil caz cel mult  $O(N^2)$ . Dacă se înlocuiește tabela hash cu un tablou sortat, se poate obține un algoritm de complexitate  $O(N^2 \log N)$ . Algoritmul propus: pentru fiecare pereche de puncte care nu se continuă cu un punct la aceeași distanță spre stânga, se numără și se rețin în tabela hash toate punctele care continuă la aceeași distanță în direcția opusă. Algoritmul are complexitatea  $O(N^2)$ : dacă o pereche se continuă cu K puncte spre dreapta, se consumă timp  $O(K)$ , dar pentru oricare K perechi de puncte consecutive enumerate acum nu vom mai face decât verificarea inițială.

### 2. a007

Problema cere de fapt să se determine dacă o mulțime dată de puncte admite un centru de simetrie. În prima etapă se determină posibilul centru de simetrie ca fiind punctul aflat la mijlocul distanței dintre punctul cel mai din stânga-jos și punctul cel mai din dreapta-sus. În etapa a doua pentru fiecare punct din mulțimea de puncte dată se adaugă la mulțime simetricul său; în final, dacă punctul ales este centru de simetrie trebuie ca fiecare punct să apară în mulțime de exact două ori (o dată citit din fișier și o dată adăugat ca simetricul unui punct). Punctele le putem împărti în N grupe utilizând o tabelă hash statică. Pentru aceasta folosim o funcție aleatoare, timpul de execuție fiind de ordinul  $O(N+C)$ . Pentru a determina un C cât mai mic se pot testa aleator funcții sau dintr-un număr prestabilit de funcții se alege aceea care furnizează c-ul minim.

## 6. Geometrie computațională

### 1. S.O.S. AEDAR098

Se determină înfășurătoarea convexă pentru fiecare mulțime de puncte. Dacă cele două poligoane obținute se intersectează, problema nu are soluție. În caz contrar, se determină o dreapta de separare posibilă (de exemplu, mediatoarea distanței dintre cele două poligoane).

### 2. Moșia lui Păcală

Se determină înfășurătoarea convexă a mulțimii parilor, apoi se calculează pentru fiecare par i aria  $d[i] * dist(par[i-1], par[i+1]) / 2$  și se determină prin programare dinamică secvența de pari nealăturăți care conduce la suma maximă a arilor.

### 3. Robot

Asociem problemei un graf (graful de vizibilitate) astfel: fiecare punct reprezintă un vîrf al grafului; între oricare două vîrfuri există muchie dacă și numai dacă cele două vîrfuri „se văd” (segmentul care le unește nu intersectează interiorul niciunui obstacol); costul unei muchii este distanța euclidiană dintre punctele corespunzătoare. Cel mai scurt drum de la punctul initial la cel final se poate obține aplicând algoritmul lui Dijkstra în graful de vizibilitate.

### 4. Galerie

Se triangulizează poligonul. Asociem triangulizării un graf astfel: vîrfurile poligonului sunt vîrfurile grafului; muchiile grafului sunt laturile poligonului și diagonalele utilizate în triangulizare. Graful asociat triangulizării este 3-colorabil. Colorăm vîrfurile grafului și alegem culoarea cel mai puțin utilizată. Plasăm câte o cameră video în fiecare vîrf colorat în culoarea selectată.

### 6. Ajutor

Se consideră mai întâi cele N puncte și se face o preprocessare. Se iau coordonatele x într-un vector și se sortează. În același mod procedăm și cu y. Fiecare punct de o coordonată din vectorul x, iar alta din vectorul y, va reprezenta un nod într-o rețea ortogonală. Pentru aceste noduri se calculează printr-o dinamică de complexitate  $O(N^2)$  distanța până la cel mai apropiat post de prim ajutor. Mai exact, se aplică programarea dinamică de patru ori: o dată începând din stânga-sus, a doua oară începând din stânga-jos, a treia oară începând din dreapta-jos și a patra oară începând din dreapta-sus. Pentru aceasta se va utiliza o matrice a (a[i][j] reprezentând distanța până la cel mai apropiat post de prim ajutor, dacă am merge numai în stânga și sus – de exemplu, pentru primul caz din poziția (i,j), adică din a i-a coordonată y, după sortare și a j-a coordonată x după sortare). Calculul se face în  $O(N^2)$ , pentru că valoarea a[i][j] nu depinde decât de vecinii din stânga și sus. Dintre toate cele patru valori se reține minimumul și se memorează într-o matrice.

Pentru fiecare dintre cele M puncte se află minimumul în  $O(1 \log N)$ , pentru că nu trebuie decât să determinăm în ce poziție ne aflăm (două căutări binare în vectorii x și y), iar apoi să determinăm în care dintre cele patru colțuri e mai bine să mergem.

Complexitate:  $O(N^2 + M \log N)$ .

## Bibliografie

1. Andonie, Răzvan ; Gârbacea, Ilie, *Algoritmi fundamentali. O perspectivă C++*, Editura Libris, Cluj, 1995.
2. Berinde, R. ; Ghinea, D. ; Ciocină, H.A. ; Margine, C., *Probleme de informatică date la concursurile internaționale*, Editura Fundației Pro, București, 2003.
3. Cerchez, Emanuela, *Informatica. Culegere de probleme pentru liceu*, Editura Polirom, Iași, 2002.
4. Cerchez, Emanuela ; Șerban, Marinel, *Programarea în limbajul C/C++ pentru liceu*. Volumele 1-2, Editura Polirom, Iași, 2004, 2005.
5. Cormen, Thomas ; Leiserson, Charles ; Rivest, Ronald, *Introduction to Algorithms*, The Massachusetts Institute of Technology, 1990.
6. Horowitz, Ellis ; Sahni, Sartaj ; Anderson-Freed, Susan, *Fundamentals of Data Structures in C.*, Computer Science Press, New York, 1993.
7. Ivașc, Cornelia ; Prună, Mona ; Mateescu-Cerchez, Emanuela, *Bazele Informaticii. Caiet de laborator*. Editura Petrion, București, 1997.
8. Jamsa, Kris, *Succes cu C++*, Editura All, București, 1997.
9. Jamsa, Kris, *- Totul despre C și C++*, Editura Teora, București, 2001.
10. Livovschi, Leon ; Georgescu, Horia, *Sinteza și analiza algoritmilor*, Editura Științifică și Enciclopedică, București, 1986.
11. Lucanu, Dorel, *Bazele proiectării programelor și algoritmilor*, Editura Universității „A.I. Cuza”, Iași, 1996.
12. Mitrana, Victor, *Provocarea algoritmilor*, Editura Agni, București, 1994.
13. Niculescu, St. ; Cerchez, Em. ; Șerban M. ș.a., *Bacalaureat și atestat în informatică*, Editura L&S Infomat București, 2000.
14. O'Rourke, Joseph, *Computational Geometry in C*, Cambridge University Press, New York, 1994.
15. Sedgewick, Robert, *Algorithms in C++*, Addison-Wesley Publishing Company, 1992.
16. S.N.E.E., Subiecte bacalaureat 2000-2006.
17. Tomescu, Ioan, *Combinatorică și teoria grafurilor*, Editura Universității București, 1978.

La Editura POLIROM

au apărut:

- Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu* (vol. I)  
Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu. Metode și tehnici de programare* (vol. II)  
Doina Hrinciuc – *C++. Probleme rezolvate și algoritmi*  
Doina Hrinciuc – *Bazele programării în C*  
Silvia Curteanu – *PC. Ghid de utilizare*  
Emanuela Cerchez, Marinel Șerban – *PC. Pas cu pas* (ediția a II-a)  
Emanuela Cerchez, Marinel Șerban – *Programarea în C/C++ pentru liceu* (vol. III)

