

**Disciplina:**

**Informatică**

**Tema:**

**Probleme de concurs pentru disciplina informatică**

**Probleme de parantezare**

**Autori:**

**Prof. Szabó Zoltan**

**Prof. Illés Ildikó**

**Material realizat pentru elevii Lotului Național de Informatică - Sovata**

**- 2014 -**

**Probleme de parantezare**

Cu această ocazie vom trata trei tipuri diferite de probleme, care reprezintă trei puncte de vedere pentru aceeaşi problemă de bază.

La început vom studia câteva probleme enunţate la diferite concursuri de programare, apoi vom prezenta câteva probleme care se bazează pe aceleași principii de programare.

Tehnicile de programare cu care ne vom întâlni sunt următoarele:

- backtracking

- programare dinamică

- recursivitate

- combinatorică

Întrucât algoritmul backtracking se predă la şcoală, nu voi intra în prezentarea detaliată a algoritmului. Acest algoritm de cele mai multe ori (în cazul nostru la toate problemele prezentate) are un timp de execuţie exponenţial, ceea ce nu este agreat de către programatori.

Şi totuşi, la concursuri încă mai pot apărea astfel de probleme. De aceea, referitor la această temă voi încerca să ating câteva aspecte, care pot duce la reducerea timpului de execuţie (să le numim „shmen-uri”).

**Problema 1:**

*Să se grupeze n perechi de paranteze în toate modurile posibile corect matematic.*

Ex. Pt. *n*=3 sunt 5 cazuri distincte:

((())) (()()) (())() ()(()) ()()()

Problemă tipică de backtracking. În stivă se vor introduce cele două tipuri de paranteze (pentru simplitate le putem codifica cu ‚(‘=0 respectiv ‚)’=1 ).

Vom avea grijă pe tot parcursul umplerii stivei, ca numărul parantezelor închise (*npi*) să nu depăşească numărul parantezelor deschise (*npd*), iar în final trebuie să avem egalitatea *npd*=*npi*.

Pentru a economisi timp la obţinerea rezultatelor, putem să folosim contorizarea parantezelor deschise, şi în momentul când am ajuns la *npd=n,* vom tipări forţat rezultatul, ştiind, că completarea secvenţei până la lungimea 2*n* conţine obligatoriu numai paranteze închise.

**Problema 2:** (Olimpiada judeţeană Târgu Mureş, Clasa a XI-a, 1994).

*Se dau 2n puncte în plan prin coordonatele carteziene (xi,yi), i=1..2n. Să se verifice dacă aceste puncte se află pe un cerc cu originea în O(0,0), apoi să se unească aceste puncte două câte două în toate modurile posibile, astfel încât oricare două segmente am lua să aibă proprietatea că nu se intersectează.*

La această problemă avem trei părţi importante:

1. verificarea dacă punctele aparţin unui cerc: (se rezolvă foarte uşor cu teorema lui Pitagora)

2. ordonarea punctelor în ordine trigonometrică sau invers trigonometrică, apoi

3. obţinerea tuturor soluţiilor cu un algoritm backtracking.

Ordonarea punctelor în plan se poate realiza după panta dreptelor OA*i*. (Această metodă de ordonare apare şi într-o altă problemă de olimpiadă judeţeană – Moşia lui Păcală-2004).

După ordonarea punctelor urmează generarea tuturor soluţiilor folosind tehnica backtracking. In stivă vom introduce numărul de ordine a punctelor, o soluţie se va obţine atunci, când în stivă s-au introdus 2*n* numere (fiecare soluţie fiind o permutare (1 .. 2*n*) numere, reprezentând segmentele cu extremităţile [*Ast[2i]Ast[2i+1]*]validate în momentul introducerii în stivă.

Ştiind că punctele sunt numerotate în ordine trigonometrică, verificarea intersecţiei a două segmente se poate rezolva fără verificarea efectivă a coordonatelor punctelor, ţinând cont numai de numărul de ordine a extremităţilor segmentelor.

Astfel două segmente [*i*,*j*] şi [*k,l*] (*i*<*j* ,*k<*l şi *i,j,k,l* distincte două câte două) se vor intersecta dacă i<*j<k<l* sau *j<i<l<k.* În celelalte cazuri cele două segmente vor fi disjuncte.

**Problema 3** (Olimpiada Judeţeană de Informatică, 2002)

*O triangulaţie a unui poligon convex este o mulţime formată din diagonale ale poligonului care nu se intersectează în interiorul poligonului ci numai în vârfuri, şi care împart toată suprafaţa poligonului în triunghiuri. Fiind dat un poligon cu n vârfuri notate 1, 2, ..., n să se genereze toate triangulaţiile distincte ale poligonului. Două triangulaţii sunt distincte dacă diferă prin cel puţin o diagonală.*

Avem un poligon convex cu *n* vârfuri, (*n* citit de la tastatură). Vârfurile poligonului sunt numerotate în ordine invers trigonometrică de la 1 la *n.* Să se obţină toate triangulările distincte posibile, respectiv numărul triangulărilor. Rezultatele se vor tipări într-un fişier astfel: pe prima linie numărul triangulărilor distincte, iar pe următoarele linii câte o triangulare înşirând perechile de puncte care reprezintă a diagonală din triangulare, scrise în ordine lexicografică.

Prin **triangulare** se înţelege împărţirea unui poligon în triunghiuri disjuncte prin unirea unor vârfuri între ele prin segmente. Dacă un poligon are n vârfuri, atunci se vor adăuga *n*-3 segmente şi se vor obţine *n*-2 triunghiuri.

Restricţie *n*<=11.

Exemplu: Pt. *n=*5 conţinutul fişierului va fi:

5 (numărul soluţiilor distincte)

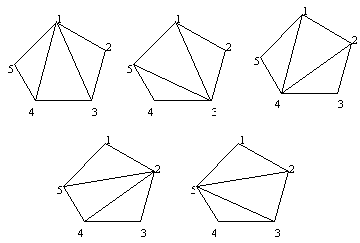
1 3 1 4

1 3 3 5

1 4 2 4

2 4 2 5

2 5 3 5



Această problemă, ca şi cea precedentă, se rezolvă cu metoda backtracking (altfel nu se pot genera toate soluţiile distincte). Însă conţine o noutate: se cere afişarea numărului de soluţii distincte înainte de tipărirea tuturor soluţiilor.

**a. Soluţia brută** ar fi, ca să lansăm algoritmul backtracking în execuţie pentru numărarea soluţiilor, şi după ce s-a obţinut acest număr ne apucăm din nou, şi vom genera cu un algoritm asemănător toate soluţiile distincte. Această rezolvare este corectă dar ineficientă.

**b. O variantă îmbunătăţită.** Dacă observăm valoarea mică a lui *n*, imediat ne putem da seama, că avem doar un număr foarte mic de cazuri distincte. Deci putem îmbunătăţi soluţia folosind tehnica preprocesării, generarea tuturor soluţiilor şi numărarea în paralel a acestora. Apoi lansăm programul în execuţie, şi notăm pe hârtie numărul de soluţii pentru *n*=3,...,11.

Apoi valorile obţinute le vom introduce ca primă instrucţiune în programul nostru astfel:

**dacă** *n=*2 **atunci** scrie 1

**altfel** **dacă** *n=*3 **atunci** scrie 1

**altfel** **dacă** *n=*4 **atunci** scrie 2

**altfel** **dacă** *n=*5 **atunci** scrie 5

**altfel** **dacă** *n=*6 **atunci** scrie 14

**altfel** **dacă** *n=*7 **atunci** scrie 42

**altfel** **dacă** *n=*8 **atunci** scrie 132

**altfel** **dacă** *n=*9 **atunci** scrie 429

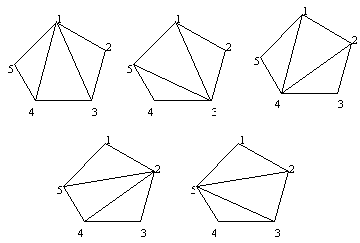
**altfel** **dacă** *n=*10 **atunci** scrie 1430

**altfel** scrie 4862 (cazul *n=*11)

După ce s-au obţinut rezultatele de mai sus, instrucţiunile pentru numărarea soluţiilor le vom elimina din programul iniţial.

**Problema 4**

*Se consideră un poligon convex cu n vârfuri. Vârfurile poligonului sunt date prin coordonate carteziene. Se cere numărul triangulărilor distincte precum şi costul triangulării optime ale acestui poligon (costul minim).* ***Costul triangulării poligonului*** *este egal cu* ***suma perimetrelor triunghiurilor*** *din care este format.*



Prin triangulare se înţelege împărţirea unui poligon în triunghiuri disjuncte prin unirea unor vârfuri între ele prin segmente. Dacă un poligon are n vârfuri, atunci se vor adăuga *n*-3 segmente şi se vor obţine *n*-2 triunghiuri.

Cerinţele s-ar putea rezolva cu metoda backtracking similar cu rezolvarea problemei 3, însă, cum nu se cere tipărirea tuturor soluţiilor, ci numai una singură, există o rezolvare mai eficientă.

Atât numărul de soluţii, cât şi triangularea optimă permită o abordare recursivă, de unde putem dezvolta un algoritm bazat pe programare dinamică.

**Cerinţa 1: Calcularea numărului de triangulări distincte:**

**1. Cu programare dinamică bazată pe o formulă recursivă**

Pentru a găsi o formulă recursivă, vom observa la început, că în poligonul cu *n* vârfuri fiacare latură aparţine numai unui singur triunghi, deci şi latura [1,*n*] aparţine unui singur triunghi, în care două vârfuri sunt 1 şi *n*, iar al treilea vârf este un vârf intermediar *k*, cu 1<*k*<*n.*

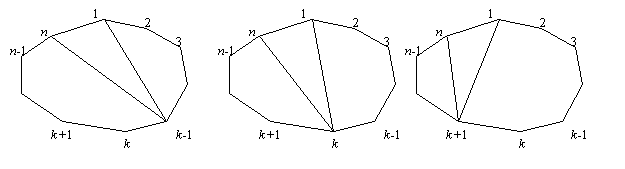
Să notăm cu *NT*(1,*n*) *numărul triangulărilor* poligonului cu *n* laturi.

Se observă, că cele *k* vârfuriintermediare pe de o parte generează cazuri disjuncte de soluţii pentru poligonul cu *n* vârfuri, şi numărul total de soluţii va fi suma acestor cazuri intermediare, iar pe de altă parte fiecare triunghi *∆1kn* desparte poligonul în două poligoane cu *k* respectiv *n-k+*1 vârfuri, care se vor rezolva analog (recursiv), iar numărul total de soluţii folosind triunghiul *∆1kn* va fi egal cu *NT*(1,*k*)\**NT*(*k*,*n*).

*NT*(1,*n*)=

Şi cum numărul de soluţii nu depinde de numărul de ordine al nodurilor, ci numai de numărul cardinal al mulţimii nodurilor vom mota *NT*(1,*n*) cu *b*(*n*), *NT*(1,*k*) cu *b*(*k*), iar *NT*(k,*n*) cu *b*(*n-k+1*), şi vom obţine o funcţie recursivă cu un singur parametru:

*b*(*n*)= , cu valoare de pornire *b*(2)=1.



Cu ajutorul unor funcţii recursive, valoarea se va calcula în timp exponenţial, dar dacă ne folosim de memoizare (adică memorarea rezoltatelor intermediare care apar de mai multe ori), calculând şi memorând elementele şirului *b*, în ordinea următoare: *b*(2), *b*(3), *... ,b*(*n-1*), *b*(*n*), atunci la fiecare pas vom folosi numai valori calculate la paşii anteriori, iar complexitatea algoritmului de calcul al lui *b*(*n*) va fi *O*(*n*2).

**subalgoritm** nr\_triang(*n*)

*b*(2)←1

**pentru** *i*←3 , *n* **execută**

*b*(*i*) ←0

**pentru** *k*←2, *i*-1 **execută**

*b*(*i*) ←*b*(*i*)+b(*k*)\**b*(*i-k+*1)

**sfpentru**

**sfpentru**

**returnează** *b*(*n*)

**sfsubalgoritm**

**2. Rezolvare prin formulă nerecursivă**

Valoarea elementului *b*(*n*) este egală cu *numărul lui Catalan* (*NC*) de ordin *n*-2,

*NC*(*n*)=, iar *b*(*n*)= *NC*(*n-*2)=.

Algoritmul care va calcula formula de mai sus va avea complexitate *O*(*n*).

**Cerinţa 2: Găsirea unei triangulări optime (cost minim)**

Considerăm varianta de la problema 4, unde o triangulare va avea costul perimetrelor triunghiurilor.

Să notăm valoarea triangulării optime a poligonului cu *n* vârfuri cu *Topt*(1,*n*).

Ca şi în cazul calculării numărului de soluţii distincte *NT*(1,*n*), şi acuma putem observa că triunghiurile *∆1kn* cu două vârfuri fixe (1 respectiv *n*) şi un vârf mobil *k, k=*2 *.. n-*1 generează cazuri disjuncte de soluţii pentru poligonul cu *n* vârfuri.

Fiecare triunghi *∆1kn* desparte poligonul în două poligoane cu *k* respectiv *n-k+1* vârfuri, optimul problemei constă în găsirea celei mai mici costuri ale optimelor locale de forma *Topt*(1,*k*)*+Topt*(*k,n*)+*Perimetru(1,k,n)*

Astfel triangularea optimă se poate obţine printr-o formulă recursivă:

*Topt*(1,*n*)=min{*Topt*(1,*k*)*+Topt*(*k,n*)+*Perimetru(1,k,n)*} cu *k*=2,..,*n-*1

folosind recursivitatea pentru orice subpoligon cu vârfurile cuprinse între *i*,*i+*1,...,*j*

*Topt*(*i*,*j*)=min{*Topt*(*i*,*k*)*+Topt*(*k,j*)+*Perimetru(i,k,j)*} cu *k*=*i+*1,..,*j-*1

Este evident că

*Topt*(*i*,*i+*1)=0 (adică două vârfuri succesive nu formează un triunghi, deci perimetrul=0), iar

*Topt*(*i*,*i+*2)= *Perimetru(i,i+*1*,i+*2*)* (trei vârfuri formează un singur triunghi, deci perimetrul este minim).

Pentru rezolvarea problemei vom avea nevoie de o matrice triunghiulară a costurilor, pe care o vom nota cu *T,* de dimensiuni *n\*n* (*Topt* are doi parametri, ambii parametri fiind un vârf al poligonului).

Să vedem cum se obţine triangularea optimă în cazul unui poligon cu 7 vârfuri:

Pasul 1. Valori iniţiale:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 |  | 0 | *P*(1,2,3) |  |  |  |  |
| 2 |  |  | 0 | *P*(2,3,4) |  |  |  |
| 3 |  |  |  | 0 | *P*(3,4,5) |  |  |
| 4 |  |  |  |  | 0 | *P*(4,5,6) |  |
| 5 |  |  |  |  |  | 0 | *P*(5,6,7) |
| 6 |  |  |  |  |  |  | 0 |
| 7 |  |  |  |  |  |  |  |

Pasul 2. Calculăm costurile optime pentru poligoanele cu 4 vârfuri

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 |  | *T*(1,2) | *T*(1,3) | *T*(1,3)+*T*(3,4)+*P*(1,3,4)  *T*(1,2)+*T*(2,4)+*P*(1,2,4) |  |  |  |
| 2 |  |  | *T*(2,3) | *T*(2,4) | *T*(2,4)+*T*(4,5)+*P*(2,4,5)  *T*(2,3)+*T*(3,5)+*P*(2,3,5) |  |  |
| 3 |  |  |  | *T*(3,4) | *T*(3,5) | *T*(3,5)+*T*(5,6)+*P*(3,5,6)  *T*(3,4)+*T*(4,6)+*P*(3,4,6) |  |
| 4 |  |  |  |  | *T*(4,5) | *T*(4,6) | *T*(4,6)+*T*(6,7)+*P*(4,6,7)  *T*(4,5)+*T*(5,7)+*P*(4,5,7) |
| 5 |  |  |  |  |  | *T*(5,6) | *T*(5,7) |
| 6 |  |  |  |  |  |  | *T*(6,7) |
| 7 |  |  |  |  |  |  |  |

Pasul 3. Calculăm costurile optime pentru poligoanele cu 5 vârfuri:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 |  | *T*(1,2) | *T*(1,3) | *T*(1,4) | *T*(1,4)+*T*(4,5)+*P*(1,4,5)  *T*(1,3)+*T*(3,5)+*P*(1,3,5)  *T*(1,2)+*T*(2,5)+*P*(1,2,5) |  |  |
| 2 |  |  | *T*(2,3) | *T*(2,4) | *T*(2,5) | *T*(2,5)+*T*(5,6)+*P*(2,5,6)  *T*(2,4)+*T*(4,6)+*P*(2,4,6)  *T*(2,3)+*T*(3,6)+*P*(2,3,6) |  |
| 3 |  |  |  | *T*(3,4) | *T*(3,5) | *T*(3,6) | *T*(3,6)+*T*(6,7)+*P*(3,6,7)  *T*(3,5)+*T*(5,7)+*P*(3,5,7)  *T*(3,4)+*T*(4,7)+*P*(3,4,7) |
| 4 |  |  |  |  | *T*(4,5) | *T*(4,6) | *T*(4,7) |
| 5 |  |  |  |  |  | *T*(5,6) | *T*(5,7) |
| 6 |  |  |  |  |  |  | *T*(6,7) |
| 7 |  |  |  |  |  |  |  |

Pasul 4. Calculăm costurile optime pentru poligoanele cu 6 vârfuri:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 |  | *T*(1,2) | *T*(1,3) | *T*(1,4) | *T*(1,5) | *T*(1,5)+*T*(5,6)+*P*(1,5,6)  *T*(1,4)+*T*(4,6)+*P*(1,4,6)  *T*(1,3)+*T*(3,6)+*P*(1,3,6)  *T*(1,2)+*T*(2,6)+*P*(1,2,6) |  |
| 2 |  |  | *T*(2,3) | *T*(2,4) | *T*(2,5) | *T*(2,6) | *T*(2,6)+*T*(6,7)+*P*(2,6,7)  *T*(2,5)+*T*(5,7)+*P*(2,5,7)  *T*(2,4)+*T*(4,7)+*P*(2,4,7)  *T*(2,3)+*T*(3,7)+*P*(2,3,7) |
| 3 |  |  |  | *T*(3,4) | *T*(3,5) | *T*(3,6) | *T*(3,7) |
| 4 |  |  |  |  | *T*(4,5) | *T*(4,6) | *T*(4,7) |
| 5 |  |  |  |  |  | *T*(5,6) | *T*(5,7) |
| 6 |  |  |  |  |  |  | *T*(6,7) |
| 7 |  |  |  |  |  |  |  |

Pasul 5. Calculăm costul optim pentru poligonul cu 7 vârfuri:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 |  | *T*(1,2) | *T*(1,3) | *T*(1,4) | *T*(1,5) | *T*(1,6) | *T*(1,6)+*T*(6,7)+*P*(1,6,7)  *T*(1,5)+*T*(5,7)+*P*(1,5,7)  *T*(1,4)+*T*(4,7)+*P*(1,4,7)  *T*(1,3)+*T*(3,7)+*P*(1,3,7)  *T*(1,2)+*T*(2,7)+*P*(1,2,7) |
| 2 |  |  | *T*(2,3) | *T*(2,4) | *T*(2,5) | *T*(2,6) | *T*(2,7) |
| 3 |  |  |  | *T*(3,4) | *T*(3,5) | *T*(3,6) | *T*(3,7) |
| 4 |  |  |  |  | *T*(4,5) | *T*(4,6) | *T*(4,7) |
| 5 |  |  |  |  |  | *T*(5,6) | *T*(5,7) |
| 6 |  |  |  |  |  |  | *T*(6,7) |
| 7 |  |  |  |  |  |  |  |

Rezultatul final se va obţine în colţul din dreapta-sus al matricei *c,* în *c*[1,*n*].

Să observăm, că după ce am dat valorile iniţiale necesare pornirii algoritmului (pasul 1), parcurgerea şi calcularea elementelor matricei costurilor se pot realiza atât oblic, cât şi vertical (paşii 2-5).

Var 1. Var 2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 |  | *T*(1,2) | *T*(1,3) |  |  |  |  | 1 |  | *T*(1,2) | *T*(1,3) |  |  |  |  |
| 2 |  |  | *T*(2,3) | *T*(2,4) |  |  |  | 2 |  |  | *T*(2,3) | *T*(2,4) |  |  |  |
| 3 |  |  |  | *T*(3,4) | *T*(3,5) |  |  | 3 |  |  |  | *T*(3,4) | *T*(3,5) |  |  |
| 4 |  |  |  |  | *T*(4,5) | *T*(4,6) |  | 4 |  |  |  |  | *T*(4,5) | *T*(4,6) |  |
| 5 |  |  |  |  |  | *T*(5,6) | *T*(5,7) | 5 |  |  |  |  |  | *T*(5,6) | *T*(5,7) |
| 6 |  |  |  |  |  |  | *T*(6,7) | 6 |  |  |  |  |  |  | *T*(6,7) |
| 7 |  |  |  |  |  |  |  | 7 |  |  |  |  |  |  |  |

Inițializarea matricei este identică atât pentru varianta 1 cât și pentru varianta 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | *i+*1 | *i*+2 | ... | *j-*1 | *j-*1 | *j* |
| *i* | *T*(*i,i+*1) | *T*(*i,i+*2) | ... | *T*(*i*,*j-*2) | *T*(*i,j-*1) | *T*(*i*,*j*) |
| *i+*1 |  |  |  |  |  | *T*(*i+*1,*j*) |
| *i+*2 |  |  |  |  |  | *T*(*i+*2,*j*) |
| ... |  |  |  |  |  | ... |
| *j-*2 |  |  |  |  |  | *T*(*j-*2,*j*) |
| *j-*1 |  |  |  |  |  | *T*(*j-*1,*j*) |

Algoritmul pentru varianta 1 (parcurgere oblică):

**subalgoritm** triangulare\_opt(*n*)

**pentru** *i*←1, *n*-2 **execută**

*t*(*i,i+2*) ←*perimetru*(*i,i+*1*,i+*2)

**sfpentru**

**pentru** *d*←3, *n*-1 **execută** (parcurgere oblică cu diferenţa constantă *d=j-i*)

**pentru** *i*←1, *n-d* **execută** (elementul de pe fiecare linie )

*min*← +∞

**pentru** *k*←2, *i+d-*1 **execută**

**dacă** *min>t*(*i,k*)+*t*(*k*, *i+d*)+*perimetru(i,k,i+d)* **atunci**

*min*←*t*(*i,k*)+*t*(*k*, *i+d*)+*perimetru(i,k,i+d)*

**sfdacă**

**sfpentru**

*t*(*i,i+d*) ←*min*

**sfpentru**

**sfpentru**

**returnează** *t*(1,*n*)

**sfsubalgoritm**

Algoritmul pentru varianta 2 (parcurgere verticală):

**subalgoritm** triangulare\_opt(*n*)

**pentru** *i*←1, *n*-2 **execută**

*t*(*i,i+2*) ←*perimetru*(*i,i+*1*,i+*2)

**sfpentru**

**pentru** *j*←4, *n* **execută** (parcurgerea fiecărei coloane)

**pentru** *i*←*j-3,*1 **pas** -1**execută** (toate liniile necalculate de la simplu spre complex)

*min*← +∞

**pentru** *k*←*j-*1, *i+*1 **pas** -1 **execută**

**dacă** *min>t*(*i,k*)+*t*(*k*, *j*)+*perimetru(i,k,j)* **atunci**

*min*←*t*(*i,k*)+*t*(*k*, *j*)+*perimetru(i,k,j)*

**sfdacă**

**sfpentru**

*t*(*i,j*) ←*min*

**sfpentru**

**sfpentru**

**returnează** *t*(1,*n*)

**sfsubalgoritm**

Atât varianta 1, cât şi varianta 2 au complexitatea *O*(*n*3).

Tabelul *t* are destule informaţii şi pentru a reconstrui triangularea optimă, adică descompunerea poligonului în triunghiuri cu suma perimetrelor minimă. Subprogramul *soluţie\_opt*(1*,n*) realizează această descompunere:

**subalgoritm** *soluţie\_opt*(*i*,*j*)

**dacă** *i=j-*2 **atunci**

*tipăreşte triunghi(i,i+*1*,i+2)*

**altfel**

**pentru** *k*←*j-*1, *i+*1 **pas** -1 **execută**

**dacă** *t*(*i,j*)*>t*(*i,k*)+*t*(*k*, *j*)+*perimetru(i,k,j)* **atunci**

*soluţie\_opt*(*i,k*)

*tipăreşte triunghi(i,k,j)*

*soluţie\_opt*(*k,j*)

*părăseşte ciclul*

**sfdacă**

**sfpentru**

**sfsubalgoritm**

**Problema 5. Nunta (Olimpiada Judeţeană de Informatică cl. XI-XII - 2002)**

*N peţitori aşezaţi la coadă, unul în spatele celuilalt. Fiecare poartă sub mantie un număr de pietre preţioase pe care doreşte să le ofere prinţesei ca dar de nuntă. Pentru a nu semăna vrajbă în rândurile lor, prinţesa a decis să-i determine ca N-1 dintre ei să renunţe în chip paşnic, peţitorul rămas devenind alesul prinţesei (indiferent de numărul de pietre preţioase deţinute de acesta).*

*Doi peţitori vecini la coadă se pot înţelege între ei astfel: cel care are mai puţine pietre preţioase pleacă de la coadă primind de la celălalt un număr de pietre astfel încât să plece acasă cu un număr dublu de pietre faţă de câte avea. Dacă doi peţitori au acelaşi număr de pietre, unul din ei (nu contează care) pleacă luând toate pietrele vecinului său.*

*Un peţitor se poate înţelege la un moment dat cu unul singur dintre cei doi vecini ai săi. După plecarea unui peţitor, toţi cei din spatele lui avansează.*

*De exemplu: pentru configuraţia alăturată de 5 peţitori, un şir posibil de negocieri care conduc la reducerea cozii la un singur peţitor este: se înţeleg vecinii 4 cu 5 şi pleacă 4, se înţeleg apoi 1 cu 2 şi pleacă 1, se înţeleg apoi 3 cu 2 şi pleacă 3, se înţeleg 2 cu 5 şi pleacă 5. Astfel peţitorul 2 câştigă mâna preafrumoasei prinţese, oferindu-i 0 pietre preţioase ca dar de nuntă.*

*Fie P numărul de pietre preţioase pe care le are peţitorul care va deveni alesul prinţesei.* ***Se cer valorile distincte ale lui P la care se poate ajunge prin toate succesiunile de negocieri posibile.***

Restricţii :

- 1  n  50,

- numărul de pietre preţioase pe care le deţin peţitorii  [0, 20].

**Exemplu:**

*nunta.in*

4

1 4 2 6

*nunta.out*

3

1 3 5

Algoritmul de rezolvare al acestei probleme se foloseşte de parantezarea în toate modurile posibile a unei expresii care conţine numai operaţii de scădere (deci este înrudită cu problema 1). Metoda de rezolvare se bazează pe un algoritm asemănător cu găsirea triangulării optime.

Apare o modificare esenţială în algoritm. Pentru că trebuiesc generate toate numerele *p* de pietre preţioase ce se pot obţine, rezultatul va fi o mulţime (deci nu un singur număr) în consecinţă elementele matricei vor fi mulţimi, iar rezultatul final va fi mulţimea de pe poziţia *a*[1*,n*].

Subalgoritmul de mai jos fiind comentat, nu necesită explicaţii suplimentare.

**subalgoritm** *peţitori*(*n*)

**pentru** *i*:=1, *n*-1 **execută**

**pentru** *j*:=*i*+1, *n* **execută** {fiecare element se intializeaza cu multimea vida}

a[*i,j*]:=[*mulţimea vidă*];

**pentru** *i*←1, *n* **execută**

*a*(*i,i*) ←[*d*(*i*)] {diagonala principala = nr de diamante a persoanei *i*}

**sfpentru**

**pentru** *j*←2, *n* **execută** (parcurgerea fiecărei coloane)

**pentru** *i*←*j-1,*1 **pas** -1**execută** (toate liniile necalculate de la simplu spre complex)

**pentru** *k*←1, *j-i* **execută**

**pentru** *l*←0, 20 **execută** {verificam toate diferentele posibile}

**pentru** *m*←0, 20 **execută**

**dacă** (*l*  a[*j-k*+1,*j*]) **şi** (m  a[*i*,*j*-*k*]) **atunci** {daca avem o combinatie posibila de }

{diamante }

a[*i,j*] ← a[*i,j*]  [abs(*l-m*)]; {atunci adaugam la multimea solutiilor}

**sfdacă**

**sfpentru**

**sfpentru**

**sfpentru**

**sfpentru**

**sfpentru**

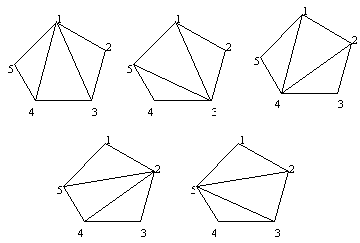
**returnează** *a*(1,*n*)

**sfsubalgoritm**

În continuare propun spre rezolvare câteva probleme echivalente

**Problema 6**

*Se consideră un poligon convex cu n vârfuri. Vârfurile poligonului sunt date prin coordonate carteziene. Se cere numărul triangulărilor distincte precum şi costul triangulării optime ale poligonului (costul minim).* ***Costul triangulării poligonului*** *este egal cu* ***suma lungimilor diagonalelor*** *care formează triangularea (fără laturile triunghiului).*



Se observă că problema 4 şi problema 6 sunt echivalente, dacă notăm cu *cm1 -* costul minim de la prima problemă, cu *cm2 -* costul minim de la problema a doua, iar cu *p -* perimetrul poligonului, atunci între cele trei entităţi avem relaţia *cm1*=*p+*2\**cm2*, iar numărul de triangulări distincte e acelaşi.

Propunere: încercaţi să rezolvaţi problema 6 cu o metodă asemănătoare problemei 4, însă fără să memoraţi perimetrele triunghiurilor, ci doar costurile diagonalelor.

**Problema 7. Parantezarea optimă a înmulţirii matricilor**

*Trebuie să efectuăm înmulţirea unui şir de matrici de dimensiuni diferite:*

*1An1,n2\* 2An2,n3\*…\* k-2Ank-2,nk-1\* k-1Ank-1,nk*

*Exponentul din stînga expresiei reprezintă numărul de ordine a matricei, iar indicii din dreapta reprezintă dimensiunea fiecărei matrici.*

*Se ştie că timpul de execuţie a înmulţirii a două matrici este determinată de operaţia de înmulţire a două elemente ale matricii, deci costul parantezării va fi direct proporţional cu numărul de înmulţiri ale numerelor reale efectuate în matrice.*

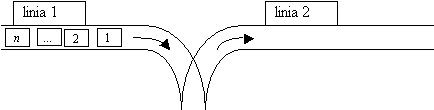
*De exemplu înmulţirea A2,5\*B5,3 va necesita 2\*3\*5=30 înmulţiri şi se va obţine ca rezultat o matrice C2,3.*

*Se cere să se parantezeze optim şirul de matrice obţinând atât numărul de înmulţiri efectuate cât şi o parantezare optimă a matricilor.*

**Problema 8**

*Se dau 2n puncte în plan care formează un poligon convex. Se cere să se unească aceste puncte două câte două, astfel încât să se obţină lungimea maximă a lungimilor totale, iar segmentele obţinute să nu se intersecteze două câte două.*

**Problema 9**



*Pe o linie ferată se găsesc n vagoane numerotate de la 1 la n. În câte moduri distincte se pot forma toate garniturile de tren pe o altă şină, ştiind că cele două şine au o parte comună, porţiune unde se pot depozita temporar oricâte vagoane.*

**Bibliografie:**

**1. Cormen, Leiserson,Rivest:** ***Introducere în algoritmi***, Computer Libris Agora, Cluj-Napoca

**2. Informatică pentru grupele de performanţă, clasa a X-a,** Dacia Educaţional, Cluj-Napoca

**3. olimpiada.info – site-ul oficial al olimpiadei de informatică**

**4. www.infoarena.ro**