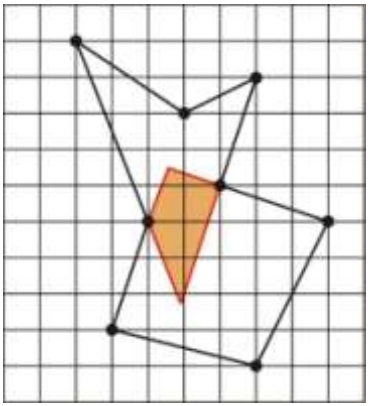
Ministerul Educației al Republicii Moldova
Universitatea de Stat din Tiraspol

Sergiu CORLAT **Anatol GREMALSCHI**

Metodologia rezolvării problemelor de geometrie computațională



Chişinău, 2013

Aprobată pentru editare de Senatul Universităţii de Stat Tiraspol

Lucrarea este destinată cadrelor didactice, activitatea cărora ţine de instruirea de performanţă la disciplinile Informatica şi Matematica, dar şi studenţilor facultăţilor de matematică şi informatică a instituţiilor de învăţământ superior din ţară, care studiază cursul de geometrie computaţională. Este de asemenea utilă elevilor pentru pregătirea către concursurile naţionale şi internaţionale de programare.

Autori:

Sergiu Corlat, lector superior universitar, UST
Anatol Gremalschi, dr. hab., profesor universitar, UTM

Recenzenţi:

Andrei Braicov, dr. conferenţiar universitar, UST
Inga Camerzan, dr. conferenţiar universitar, UnAŞM

519.17+004.421.2(075.8)
C71

Cuprins

Introducere 5

1. Transformări de coordonate în plan 7
   1.1 Deplasări şi rotaţii 7
   1.2. Coordonate polare 9
   1.3. Implementări 10

2. Intersecţii 12
   2.1. Intersecţia dreptelor 13
   2.2. Intersecţia dreptei cu un segment 14
   2.3. Intersecţia segmentelor 17

3. Înfăşurătoarea convexă 18
   3.1. Algoritmul elementar 18
   3.2 Algoritmul Graham. 21
   3.3 Modificarea Andrew 23

4. Triangularizări 27
   4.1. Un algoritm greedy pentru mulţimi de puncte 27
   4.2. Triangularizarea poligoanelor convexe 29
   4.3. Triangularizarea poligoanelor simple. 30

5. Apropiere şi apartenenţă 32
5.1. Cea mai apropiată pereche de puncte 32
5.2. Poligonul şi diagrama Voronoi 37
5.3. Apartenenţa punctului la un domeniu 42
5.4. Nuclee 47

6. Probleme geometrice de concurs 50
6.1 Robot 50
6.2.Robot II 52
6.3. Piatra 54
6.4 Carcase 56
6.5. Turnuri 58
6.6 Atac 59
6.7. Evadare 63
6.8. Arcaşi 64
6.9. Cetate 70
6.10. Druizi 71
6.11 Segmente 76
6.12 Sticla 81
6.13 Biliard 90

Introducere

Geometria computațională reprezintă una din ramurile importante ale matematicii aplicate moderne. Pornind de la cercetarea unor elemente geometrice simple (punct, segment, dreaptă), se ajunge la o modelare complexă a figurilor geometrice în plan şi a corpurilor în spațiul tridimensional.

Algoritmii geometriei computaționale stau la baza tuturor aplicațiilor de proiectare şi modelare grafică (aplicații CAD, procesoare de grafică vectorială, aplicații de modelare 3D). Fără ei ar fi imposibilă proiectarea construcțiilor arhitecturale moderne, realizarea proiectelor GPS, apariția majorității absolute a produselor cinematografice de succes din ultimii ani. Enumerarea domeniilor de aplicare poate fi continuată la nesfârşit.

Domeniul vast de aplicare şi multitudinea fenomenelor şi situațiilor reale, descrise în termenii geometriei computaționale, au dus la includerea problemelor geometrice în cadrul concursurilor de programare de cele mai diferite nivele.

Lucrare de față este concepută ca un curs de inițiere în algoritmii de geometrie computațională, destinat studenților facultăților de profil, pentru profesorii de informatică, precum şi pentru elevii claselor liceale, participanți la concursurile naționale şi internaționale de programare.

Un alt scop al lucrării constă în formarea abilităților de analiză şi proiectare a algoritmilor. În acest sens, algoritmii şi soluțiile problemelor propuse în lucrare sunt însoțite de descrieri ale structurilor de date, fragmente de cod, estimări ale complexității.

Cu toate că aparatul matematic necesar pentru rezolvarea problemelor de geometrie computațională este relativ simplu, implementare acestuia impune necesitatea cercetării unui număr considerabil de cazuri speciale. Acesta este un motiv, pentru care

problemele de geometrie computațională se consideră printre cele mai dificile, în special în condiții de concurs.

Cercetarea şi optimizarea soluțiilor propuse pentru principalele probleme prezente în lucrare va permite cititorului să acumuleze experiența necesară pentru rezolvare de probleme, şi, nu în ultimul rând, pentru generarea testelor destinate verificării programelor elaborate.

La baza lucrării se află o serie de articole publicate pe parcursul ultimilor ani în revista de matematică şi informatică „*Delta*", materiale şi probleme propuse în cadrul şcolilor de vară la informatică (edițiile anilor 2001–2012), precum şi probleme propuse în cadrul Concursului de pregătire continuă la informatică „.campion" (campion.edu.ro).

Ținem să aducem sincere mulțumiri tuturor celor care au contribuit la apariția acestei cărți, în special colectivului Catedrei de Informatică şi Tehnologii Informaționale a Universității de Stat din Tiraspol.

August 2013 *Sergiu Corlat, Anatol Gremalschi*

1. Transformări de coordonate în plan

1.1 Deplasări şi rotaţii

Rezolvarea oricărei probleme de geometrie computațională începe (dacă e necesar) cu deplasarea coordonatelor elementelor cercetate (de cele mai dese ori ale punctelor) într-un domeniu potrivit al sistemului de coordonate (de obicei, în cadranul unu). În unele cazuri, deplasarea poate fi însoțită şi de rotația sistemului de coordonate.

Fie într-un sistem cartezian de coordonate un punct p de coordonate (x, y). Deplasarea de coordonate de-a lungul axei Ox cu o valoare dată **a** implică o modificare a coordonatelor punctului conform formulelor:

$$\begin{cases} x' = x + a, \\ y' = y. \end{cases}$$

În cazul deplasării de-a lungul axei Oy cu valoarea **b**, transformarea de coordonate va fi dată de formulele:

$$\begin{cases} x' = x, \\ y' = y + b. \end{cases}$$

Modificarea „combinată” a coordonatelor cu **a** după axa Ox şi **b** după Oy va fi dată de formulele:

$$\begin{cases} x' = x + a, \\ y' = y + b. \end{cases}$$

Următorul tip de transformare este rotația cu un unghi φ a punctului (punctelor) față de originea sistemului de coordonate (fig. 1.1).

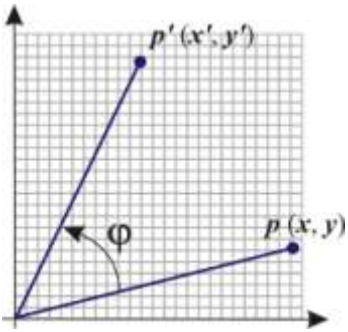


Fig. 1.1. Rotaţia punctului cu un unghi φ.

În unele cazuri, operația poate fi realizată invers: prin rotația originii sistemului de coordonate față de un punct (sau sistem de puncte) dat.

În acest caz există patru numere **a,b,c,d**, care permit de a trece univoc coordonatele (x, y) în (x', y'), utilizând sistemul de ecuații: $\begin{cases} x' = ax + by, \\ y' = cx + dy. \end{cases}$ (1.1)

Pentru determinarea acestui cuadruplu de numere pot fi folosite punctele (**1,0**) şi (**0,1**).

Se observă că la rotația cu un unghi φ punctul de coordonate (1,0) trece în punctul $(\cos\varphi, \sin\varphi)$, iar (0, 1) – în $(-\sin\varphi, \cos\varphi)$ (fig. 1.2).

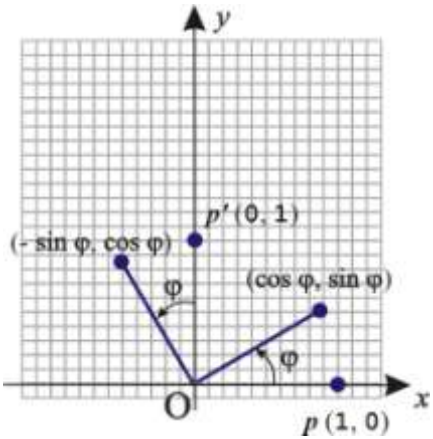


Fig. 1.2. Determinarea cuadruplului a, b, c, d, folosind punctele (1,0) şi (0,1)

După înlocuirea acestor valori în calitate de coeficienți ai sistemului (1.1) se obține:

$$\begin{cases} x' = a \\ y' = c \end{cases} \Rightarrow \begin{cases} a = \cos\varphi, \\ c = \sin\varphi. \end{cases}$$

Analog:

$$\begin{cases} x' = b \\ y' = d \end{cases} \Rightarrow \begin{cases} b = -\sin\varphi, \\ d = \cos\varphi. \end{cases}$$

Astfel, sistemul de ecuații pentru rotația cu un unghi φ a unui punct arbitrar în jurul originii sistemului de coordonate ia forma:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi \\ y' = x\sin\varphi + y\cos\varphi \end{cases}$$

În cazul, în care rotația cu unghiul φ a punctului de coordonate (x, y) este efectuată față de punctul de coordonate (x_0, y_0), diferit de originea sistemului de coordonate, mai întâi se transferă originea sistemului de coordonate în centrul de rotație (deplasarea de coordonate), apoi se efectuează rotația în jurul noii „origini a sistemului de coordonate":

$$\begin{cases} x' - x_0 = (x - x_0)\cos\varphi - (y - y_0)\sin\varphi \\ y' - y_0 = (x - x_0)\sin\varphi + (y - y_0)\cos\varphi \end{cases}$$

sau

$$\begin{cases} x' = x_0 + (x - x_0)\cos\varphi - (y - y_0)\sin\varphi \\ y' = y_0 + (x - x_0)\sin\varphi + (y - y_0)\cos\varphi \end{cases}$$

1.2. Coordonate polare

Problemele în care apare procedura de parcurgere a unei mulțimi de puncte după unghiurile formate de acestea cu axa O*x* se rezolvă mai simplu în cazul trecerii la coordonate polare. În acest sistem de coordonate fiecare punct *p* este determinat de perechea (*r*, *φ*), unde *r* reprezintă distanța de la punct până la originea sistemului de coordonate, iar *φ* - unghiul format de vectorul $\overrightarrow{op}$ cu axa O*x*.

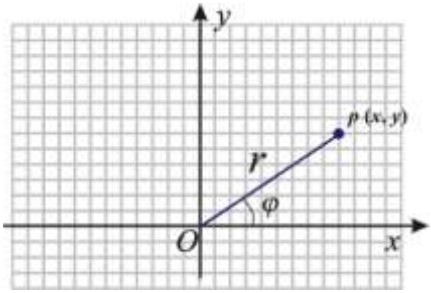


Fig. 1.3. Coordonatele polare ale punctului de coordonate carteziene (x, y)

Trecerea de la coordonatele carteziene (*x, y*) la cele polare (fig. 1.3) este determinată de formulele [1, p. 77]:

$$r=\sqrt{x^2+y^2},\qquad \varphi=\begin{cases}\arctan\dfrac{x}{y} & y>0\\ \pi+\arctan\dfrac{x}{y} & y<0\\ \dfrac{\pi}{2} & x=0,\ y>0\\ \dfrac{3\pi}{2} & x=0,\ y<0\end{cases}$$

1.3. Implementări

În problemele de geometrie computațională pot apărea diferite modele de transformare a coordonatelor: deplasări după o axă, deplasări combinate (după ambele axe), rotații față de un punct, deplasări însoțite de rotații. Toate aceste transformări presupun recalcularea coordonatelor unui punct sau ale unui sistem de puncte.

Pentru rezolvarea eficientă a problemelor de geometrie este foarte important de a alege corect structurile de date. Pentru implementarea transformărilor de coordonate este necesară descrierea unei singure structuri – punctul:

```
type point=record
           x,y : real;
     end;
```

Orice obiect geometric, definit printr-un set de puncte poate fi descris cu ajutorul unui tablou unidimensional sau o listă alocată dinamic, cu elemente de tip `point`.

Procedura de deplasare a coordonatelor cu valoarea **vx** după axa *Ox* şi **vy** după *Oy* poate fi realizată în felul următor:

```
procedure move(var P:point; vx,vy:real);
  begin
    P.x:=P.x+vx;
    P.y:=P.y+vy;
  end;
```

La fel de simplă este şi o posibilă implementare a rotației cu un unghi **u** față de un punct de coordonate **vx** şi **vy**:

```
procedure rotate (var P:point; u,vx,vy:real);
var
       old:point;
 begin
       old:=P;
       P.x:=vx+(old.x-vx)*cos(u*pi/180)
                   -(old.y-vy)*sin(u*pi/180);
       P.y:=vy +(old.x-vx)*sin(u*pi/180)
                   +(old.y-vy)*cos(u*pi/180);
 end;
```

2. Intersecţii

În majoritatea problemelor de geometrie computaţională apare în calitate de subproblemă determinarea coordonatelor punctului de intersecţie a două drepte, a două segmente sau a unui segment şi unei drepte.

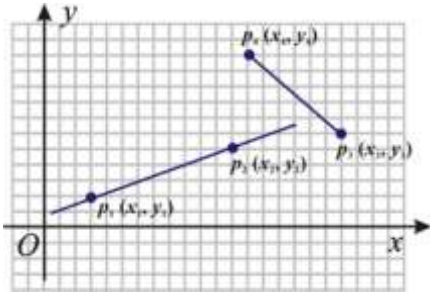


Fig. 2.1 Definirea dreptei (segmentului) prin două puncte

Metoda folosită pentru rezolvarea acestor subprobleme este aceeaşi, cu diferenţe puţin semnificative pentru fiecare caz.

Atât pentru definirea dreptei, cât şi pentru definirea segmentului se vor folosi două puncte distincte: arbitrare (ale dreptei sau extreme pentru segment, fig. 2.1).

Prin urmare, atât descrierea dreptei, cât şi descrierea segmentului pot fi realizate de aceeaşi structură de date, care va conţine coordonatele a două puncte şi, suplimentar, coeficienţii **A,B,C** ai ecuaţiei generale a dreptei. Aceşti coeficienţi vor fi necesari pentru calculul coordonatelor punctului de intersecţie. Una din posibilele metode de definire a acestei structuri este:

```
type line=record
             x1,y1,x2,y2 : real;
             A, B, C : real;
     end;
```

2.1. Intersecţia dreptelor

Ecuația generală a dreptei

Fie dreapta l determinată de punctele $p_1\,(x_1, y_1)$ şi $p_2\ (x_2, y_2)$.

Ecuația dreptei ce trece prin două puncte date este:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} \quad (2.1)$$

Din această ecuație pot fi calculați coeficienții ecuației generale a dreptei: $Ax + By + C = 0$.

Efectuând transformări elementare, din (2.1) se obține:

$$x(y_2 - y_1) + y(x_1 - x_2) + (x_2 y_1 - x_1 y_2) = 0. \quad (2.2)$$

Din (2.2) rezultă formulele de calcul pentru coeficienții ecuației generale a dreptei:

$$\begin{aligned} A &= y_2 - y_1, \\ B &= x_1 - x_2, \\ C &= x_2 y_1 - x_1 y_2. \end{aligned} \quad (2.3)$$

Determinarea punctului de intersecție a două drepte

Fie dreptele p şi l, determinate respectiv de punctele p_1 (x_1, y_1), $p_2\,(x_2, y_2)$ şi $p_3\,(x_3, y_3)$, $p_4\,(x_4, y_4)$

Determinarea punctului q de intersecție a acestor drepte se reduce la rezolvarea sistemului de ecuații:

$$\begin{cases} A_1 x + B_1 y + C_1 = 0 \\ A_2 x + B_2 y + C_2 = 0, \end{cases}$$

unde $A_1 x + B_1 y + C_1 = 0$ este ecuația generală a dreptei p şi $A_2 x + B_2 y + C_2 = 0$ este ecuația generală a dreptei l. Coeficienții acestor ecuații pot fi calculați conform formulelor (2.3).

Până la rezolvarea nemijlocită a sistemului urmează să se verifice dacă dreptele p şi l sunt paralele sau coincid. Pentru aceasta se verifică condiţiile:

- $A_1B_2 = A_2B_1$ & $A_1C_2 \neq A_2C_1$ – dreptele p şi l sunt paralele;
- $A_1B_2 = A_2B_1$ & $A_1C_2 = A_2C_1$ – dreptele p şi l coincid.

Dacă niciuna din condiţiile precedente nu se satisface, atunci există un punct unic de intersecţie a dreptelor p şi l, ale cărui coordonate pot fi calculate după formulele:

a) Dacă $A_1 \neq 0$.

$$y_{sol} = \frac{A_2C_1 - C_2A_1}{B_2A_1 - B_1A_2}, \qquad (2.4\text{ a})$$
$$x_{sol} = \frac{-B_1y_{sol} - C_1}{A_1};$$

b) Dacă $A_1 = 0$

$$y_{sol} = -\frac{C_1}{B_1}, \qquad (2.4\text{ b})$$
$$x_{sol} = -\frac{B_2y_{sol} + C_2}{A_2}.$$

Cu ajutorul setului de formule (2.4) pot fi calculate coordonatele punctului de intersecţie $q(x_{sol}, y_{sol})$ a dreptelor p şi l.

2.2. Intersecţia dreptei cu un segment

Formulele deduse în paragraful precedent permit să se calculeze coordonatele punctului de intersecţie a două drepte. Cazul intersecţiei a două segmente sau a unei drepte şi a unui segment pot fi reduse la cel al intersecţiei dreptelor, dar cu verificarea anumitor condiţii suplimentare.

Fie dreapta l care trece prin punctele p_1 şi p_2 şi segmentul s cu punctele extreme s_1 şi s_2 (fig 2.2).

Se observă, că în cazul intersecției dreptei l şi a segmentului s, extremitățile segmentului sunt poziționate de părți diferite față de vectorul $\overrightarrow{p_2p_1}$. Dacă obiectele cercetate nu se intersectează, atunci ambele extremități ale segmentului se află de aceeaşi parte a vectorului $\overrightarrow{p_2p_1}$.

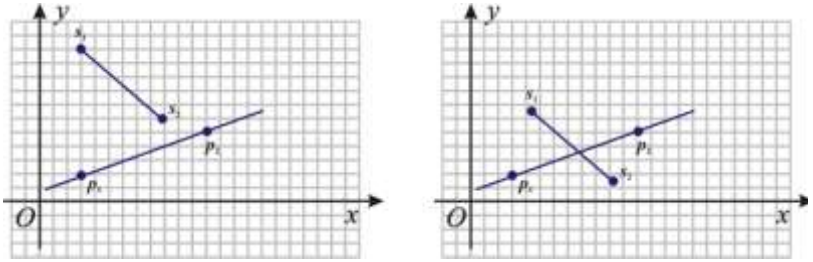


Fig. 2.2. Poziţia segmentului faţă de dreapta.

Poziţia punctului faţă de un vector

Fie vectorul $\overrightarrow{p_2p_1}$ coordonate (x_2, y_2), (x_1, y_1), şi punctul s de coordonate (x_3, y_3).

Pentru a poziționa punctul s față de vectorul $\overrightarrow{p_2p_1}$, poate fi folosit următorul determinant [12, p. 60]:

$$\Delta = \begin{vmatrix} x_2 & y_2 & 1 \\ x_1 & y_1 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

Determinantul este pozitiv dacă punctul s este situat în semiplanul drept față de vectorul $\overrightarrow{p_2p_1}$; este nul, dacă punctul s aparține dreptei determinate de acest vector, şi negativ, dacă s este plasat în semiplanul stâng.

O realizare posibilă a funcției de calcul al determinantului este următoarea:

```
function sarrus(p1,p2,p3:point[1]): real;
 begin
    sarrus:= p1.x*p2.y+p2.x*p3.y+p1.y*p3.x
             -p3.x*p2.y-p3.y*p1.x-p1.y*p2.x;
 end;
```

Intersecţia

Expresia `Sarrus(p2,p1,s1)*Sarrus(p2,p1,s2)` va avea valoare *pozitivă,* dacă dreapta l şi segmentul s nu se intersectează (ambele extremităţi ale segmentului sunt situate de aceeaşi parte a dreptei, valorile determinantului sunt de acelaşi semn), *nulă,* dacă cel puţin una dintre extremităţi aparţine dreptei (pentru extremitatea segmentului, care aparţine dreptei determinantul este egal cu `0`), *negativă,* dacă dreapta l şi segmentul s se intersectează (extremităţile segmentului sunt situate de părţi diferite ale vectorului, valorile determinantului au semne diferite).

Prin urmare, dacă pentru dreapta l, definită prin punctele p_1 şi p_2 şi segmentul s cu extremităţile s_1 şi s_2, expresia `Sarrus(p2,p1,s1)*Sarrus(p2,p1,s2)` are valoare negativă, atunci dreapta l şi segmentul s se intersectează, iar coordonatele punctului de intersecţie pot fi calculate conform formulelor 2.4.

Dacă valoarea expresiei este nulă, se verifică nemijlocit care dintre extremităţile segmentului aparţine dreptei. În cazul în care ambele extremităţi ale segmentului aparţin dreptei, tot segmentul aparţine acesteia.

[1] `type point=record  x,y : real; end;`

2.3. Intersecţia segmentelor

Fie segmente *s* şi *p* cu extremităţile s_1 s_2 , respectiv p_1 , p_2. Pentru simplitate se va considera că segmentele nu aparţin aceleiaşi drepte şi nu sunt paralele (cazurile date pot fi verificate cu ajutorul precondiţiilor pentru formulele 2.4).

Segmentele se vor intersecta numai dacă

`Sarrus(p2,p1,s1)*Sarrus(p2,p1,s2) ≤ 0` şi

`Sarrus(s2,s1,p1)*Sarrus(s2,s1,p2) ≤ 0`

Coordonatele punctului de intersecţie şi în acest caz pot fi calculate folosind formulele 2.4.

3. Înfăşurătoarea convexă

Problema determinării înfăşurătoarei convexe este una dintre problemele centrale ale geometriei computaționale. Ea apare atât în cadrul unor aplicații economice, financiare, ecologice, arhitecturale, cât şi în probleme geometrice analitice.

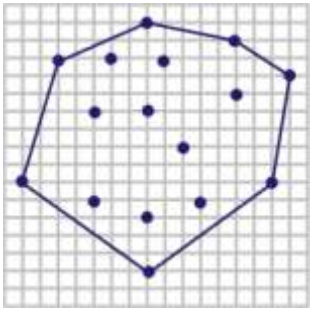


Fig. 3.1. Înfăşurătoarea convexă a unei mulţimi de puncte

Noțiunea de înfăşurătoare convexă în plan este intuitiv simplă: pentru o mulțime S de puncte ale planului, înfăşurătoarea convexă $Q(S)$ este mulțimea de vârfuri ale poligonului convex[2] cu cea mai mică arie, care conține toate punctele mulțimii S. Înfăşurătoarea convexă poate fi modelată cu ajutorul inei benzi elastice, întinse în jurul mulțimii S. La eliberare, banda elastică va repeta conturul înfăşurătoarei convexe (fig. 3.1).

3.1. Algoritmul elementar

Fie mulțimea de puncte din plan $S = \{p_1, \dots, p_N\}$. Fiecare element p_i al mulțimii este descris prin coordonatele sale carteziene (x_i, y_i). O metodă intuitivă de determinare a înfăşurătoarii convexe presupune eliminarea din mulțimea inițială a tuturor punctelor ei interioare.

Algoritmul se bazează pe două afirmații simple:

[2] Poligonul P - convex, dacă $\forall x_1, x_2 \in P$, $[x_1, x_2] \in P$.

a) Înfăşurătoarea convexă a unei mulțimi de puncte S este formată din punctele extreme ale mulțimii S;

b) un punct $p \in S$ **nu** este un punct extrem dacă şi numai dacă, există cel puțin un triunghi, determinat de punctele $p_i, p_j, p_k \in S$, astfel încât p să se afle în interiorul lui (fig 3.2).

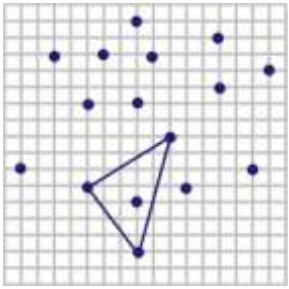


Fig. 3.2. Determinarea unui punct interior

În baza acestor afirmații, punctele interioare ale mulțimii se exclud prin cercetarea apartenenței fiecărui punct fiecărui triunghi generat de punctele mulțimii S. Atât pseudocodul, cât şi implementarea algoritmului sunt extrem de simple:

Implementarea în pseudocod

Pas 1 $C \leftarrow S$

Pas 2 Pentru toate punctele $p_i \in S$

Pentru toate punctele $p_j \in S, p_j \neq p_i$

Pentru toate punctele $p_k \in S, p_k \neq p_i, p_k \neq p_j$

Pentru toate punctele $p \in S, p \neq p_i, p \neq p_j, p \neq p_k$

dacă $p \in \Delta p_i p_j p_k$ [3] **atunci** $C \leftarrow C / \{p\}$

Apartenența punctului la un triunghi

Condiția $p \in \Delta p_i p_j p_k$ poate fi verificată prin determinarea poziției punctului p față de fiecare din vectorii $\overrightarrow{p_i p_j}$, $\overrightarrow{p_j p_k}$, $\overrightarrow{p_k p_i}$.

[3] Aici şi în continuare notația $p \in \Delta p_i p_j p_k$ va specifica triunghiul în reuniune cu domeniul său interior.

Dacă semnul valorilor returnate la apelurile funcției `sarrus(pi,pj,p)`, `sarrus(pj,pk,p)`, `sarrus(pk,pi,p)`, este același, atunci $p \in \Delta p_i p_j p_k$. Prin urmare, verificarea dacă un punct aparține interiorului unui triunghi poate fi realizată într-un număr de operații mărginit de o constantă.

Instrucțiunile ciclice din pasul 2 al algoritmului generează un număr de operații, proporțional cu N^4, ceea ce determină și complexitatea finală a algoritmului – $O(N^4)$.

Implementare

Structura de date utilizată pentru determinarea înfășurătoarei convexe este un tablou de elemente, fiecare dintre ele fiind un articol cu trei componente: coordonatele punctului și marcajul (0 – punct extrem, 1 – punct interior), care specifică apartenența la înfășurătoarea convexă:

```
type point=record
   x,y,int: integer;
   end;
```

Verificarea dacă punctul aparține la un triunghi este realizată în cadrul funcției **apart**:

```
function apart(l,i,j,k:integer) : boolean;
  var k1,k2,k3: real;
  begin
   apart:=true;
   k1:=sarrus(a[i],a[j],a[l]);
   k2:=sarrus(a[j],a[k],a[l]);
   k3:=sarrus(a[k],a[i],a[l]);
   if (k1*k2 <0 ) or (k1*k3<0) or (k2*k3<0)
       then apart:=false;
  end;
```

Aplicarea marcajelor este realizată în următorul fragment:

```
for i:=1 to n-2 do
 for j:=i+1 to n-1 do
   for k:=j+1 to n do
     for l:=1 to n do
```

```
if (l<>i) and (l<>j) and (l<>k) then
  if apart(l,i,j,k) then a[l].int:=1;
```

Afişarea coordonatelor punctelor, care formează înfăşurătoarea se realizează prin verificarea marcajelor:

```
for i:=1 to n do if a[i].int=0
            then writeln(a[i].x, ' ',a[i].y);
```

Algoritmul descris, deşi este unul polinomial, nu este cel mai eficient pentru determinarea înfăşurătoarei convexe a unei mulțimi de puncte.

3.2 Algoritmul Graham.

Un algoritm eficient cu complexitatea $O(N \log N)$ a fost propus de R. L. Graham în 1972. Algoritmul se bazează pe determinarea unui punct interior al mulțimii S, deplasarea în el a originii sistemului de coordonate, şi sortarea celorlalte puncte ale mulțimii după unghiul format de ele cu axa Ox. După sortare, punctele din S sunt plasate într-o listă bidirecțională, circulară. La următoarea etapă se parcurge lista formată, pornind de la punctul cu abscisa minimă (fig. 3.3). Acesta este un punct care, în mod cert, aparține înfăşurătoarei convexe. La fiecare „moment" al parcurgerii se cercetează un triplet de elemente[4] consecutive ale listei p_1, p_2, p_3. Cercetarea

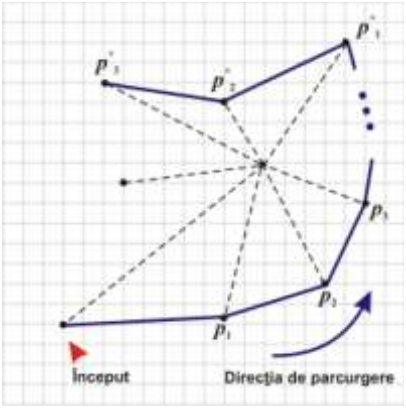


Fig. 3.3. Parcurgerea listei de puncte conform algoritmului Graham. Punctele p_1, p_2, p_3 marchează tripletul curent.

[4] Fiecare element al listei descrie un punct al mulțimii S.

este realizată prin verificarea poziției punctului p_2 față de vectorul $\overrightarrow{p_1p_3}$.

Poziționarea în semiplanul stâng stabileşte p_2 ca fiind un punct interior al mulțimii. În acest caz, p_2 este exclus din listă, iar tripletul care urmează să fie cercetat devine p_0, p_1, p_3[5].

Poziționarea în semiplanul drept stabileşte p_2 ca fiind un punct posibil al înfăşurătoarei convexe. În acest caz, p_2 este păstrat în listă, iar tripletul care urmează să fie cercetat devine p_2, p_3, p_4[6]. Parcurgerea ia sfârşit când se revine în punctul de unde a început.

Implementarea în pseudocod

Pas 1 Se determină un punct interior z, $z \in S$.

Pas 2 Se transferă originea sistemului de coordonate în punctul z.

Pas 3 Se determină coordonatele polare (r, φ) ale fiecărui punct $p \in S$, $p \neq z$, apoi se sortează după creşterea φ (pentru punctele cu unghiuri egale sortarea se efectuează după creşterea r).

Pas 4 Se formează o listă bidirecțională, circulară, ale cărei elemente sunt punctele sortate (Q).

Pas 5 Se stabileşte p_0 – punctul de abscisă minimă (în sistemul cartezian de coordonate). $p \leftarrow p_0$

Pas 6 **Cît timp** la p_0 nu se ajunge prin mişcări „înainte”, **se repetă**:

---

[5] p_0 – elementul precedent pentru p_1

[6] p_4 – elementul următor pentru p_3

a) Se consideră tripletul $p_1 \leftarrow p,\ p_2 \leftarrow p[urm]$, $p_3 \leftarrow p_2[urm]$

b) **Dacă** p_2 e poziționat în semiplanul drept față de vectorul $\overrightarrow{p_1p_3}$ atunci se efectuiază mişcarea „înainte” $p \leftarrow p[urm]$, altfel p_2 se exclude din lista Q şi se efectuiază mişcare „înapoi” $p \leftarrow p[prec]$

Pas 7 Q – înfăşurătoarea convexă.

Complexitatea algoritmului este $O(N \log N)$ şi e determinată de complexitatea pasului 3– sortarea punctelor după unghiul φ. Paşii 1, 2, 4, 5 au o complexitate liniară. Aceeaşi complexitate o are şi pasul 6 – la fiecare „moment” fie este eliminat un punct, fie se realizează un pas înainte. Numărul de operații în cadrul verificării poziției p_2 este mărginit de o constantă.

3.3 Modificarea Andrew

Modificarea algoritmului are drept scop omiterea determinării punctului intern z, deplasării originii sistemului de coordonate, şi a calculului coordonatelor polare. La baza variantei Andrew stă următorul principiu: partea superioară (după y) a înfăşurătoarei convexe este bombată (în sus) pentru oricare 3 puncte consecutive ale sale, cea inferioară – bobmbată în jos (fig 3.4).

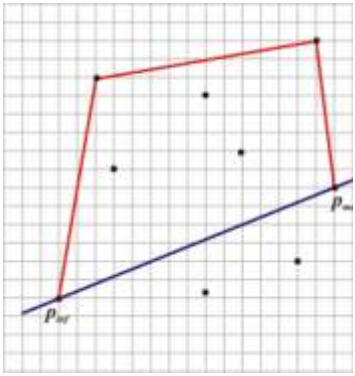


Fig. 3.4. Construirea înfăşurătoarei convexe. Modificarea Andrew

Pseudocodul algoritmului are următoarea formă:

Pas 1 Se determină două puncte $p_{\min}, p_{\max} \in S$ de abscisă minimă (respectiv maximă).

Pas 2 Se separă S în $S_{\sup}$ şi $S_{\inf}$ după poziția punctelor față de $\overrightarrow{p_{\min} p_{\max}}$. $S_{\sup}$ va fi format din punctele extreme şi cele din stânga vectorului, $S_{\inf}$ – din punctele extreme şi cele din dreapta vectorului.

Pas 3 Se sortează $S_{\sup}$, $S_{\inf}$ după creşterea abscisei.

Pas 4 **a.** Se verifică toate tripletele consecutive $p_i, p_{i+1}, p_{i+2} \in S_{\sup}$, pornind de la $p_{\min}$.

Dacă p_{i+1} este poziționat în stânga $\overrightarrow{p_i p_{i+2}}$ **atunci** se execută mişcarea „înainte”, **altfel** – mişcarea „înapoi”. La atingerea $p_{\max}$, punctele rămase în $S_{\sup}$ formează partea superioară a înfăşurătoarei convexe.

b. se verifică toate tripletele consecutive $p_i, p_{i+1}, p_{i+2} \in S_{\inf}$, pornind de la $p_{\min}$.

Dacă p_{i+1} este poziționat în dreapta $\overrightarrow{p_i p_{i+2}}$, **atunci** se execută mişcarea „înainte”, **altfel** – mişcarea „înapoi”. La atingerea $p_{\max}$, punctele rămase în $S_{\inf}$ formează partea inferioară a înfăşurătoarei convexe.

Pas 5 $Q \leftarrow S_{\inf} \cup S_{\sup}$

Mai jos este propusă o realizare a acestui algoritm. Se presupune că punctele sunt date prin coordonatele lor – numere întregi. Sortarea este realizată de procedura **qsort**, care nu este descrisă aici, fiind considerată elementară. Poziția reciprocă a punctelor este determinată de funcția **sarrus**, descrisă anterior.

Structurile de date:

drag – **mulţimea** S

sus, jos – **mulţimile** S_{sup} , S_{inf}

n – $|S|$

```
procedure conv;
var    minx,maxx:longint;
       j, imin, imax,i, nsus, njos: integer;
       rem: boolean;
       p1,p2,p3:nod;
begin
{1. determinarea extremelor dupa x}
minx:=drag[1].x; maxx:=drag[1].x;imin:=1; imax:=1;
       for i:=2 to n do
           if drag[i].x< minx then
              begin minx:=drag[i].x; imin:=i; end
           else if drag[i].x>maxx then
              begin maxx:=drag[i].x; imax:=i; end;
{2. separarea in submultimi}
nsus:=1; njos:=1; sus[1]:=drag[imin];
jos[1]:=drag[imin];
for i:=1 to n do
  if not (i in [imin, imax])then
       if sarrus(drag[imin],drag[imax],drag[i])<0
            then
             begin inc(njos); jos[njos]:=drag[i]; end
            else
             begin inc(nsus); sus[nsus]:=drag[i]; end;
inc(nsus);sus[nsus]:=drag[imax];
inc(njos);jos[njos]:=drag[imax];
{3. sortarea subseturilor }
qsort(sus,2,nsus-1);
qsort(jos,2,njos-1);
{crearea infăşurătoarei convexe}
{4. sus}
repeat
       rem:=false; i:=2;
```

```
      while i<nsus do
          begin
            p1:=sus[i-1]; p2:=sus[i]; p3:=sus[i+1];
            if sarrus(p1,p3,p2)>0 then  i:=i+1
                else begin
                       rem:=true;
                       for j:=i to nsus-1 do
                                   sus[j]:=sus[j+1];
                            dec(nsus);
                     end;
          end;
until not rem;
{si jos}
repeat
      rem:=false; i:=2;
      while i<njos do
          begin
            p1:=jos[i-1]; p2:=jos[i]; p3:=jos[i+1];
            if sarrus(p1,p3,p2)<0 then  i:=i+1
                else begin
                       rem:=true;
                       for j:=i to njos-1 do
                                 jos[j]:=jos[j+1];
                       dec(njos);
                     end;
          end;
until not rem;
{5. asamblarea}
       for i:= nsus-1 downto 2 do
         begin
              inc(njos);
              jos[njos]:=sus[i];
         end;
         drag:=jos; n:=njos;
end;{conv}
```

La finalul execuției procedurii punctele care formează înfășurătoarea convexă vor fi stocate în structura **jos**.

4. Triangularizări

Din punct de vedere geometric, triangularizarea $T(S)$ a mulțimii de puncte S este o divizare a înfășurătoarei convexe $Q(S)$ în fețe din exact 3 muchii. Suplimentar, se vor respecta condițiile:

a) vârfuri ale muchiilor pot fi numai puncte din S;
b) toate punctele mulțimii S vor fi utilizate în calitate de vârfuri. (fig. 4.1).

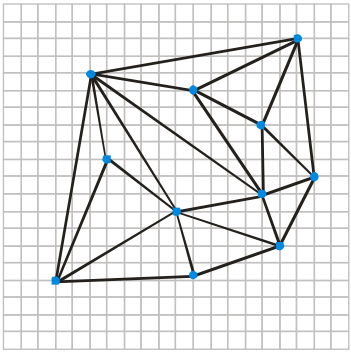


Fig. 4.1. Triangularizarea unei mulţimi de puncte

4.1. Un algoritm greedy pentru mulţimi de puncte

Fie mulțimea de puncte S şi $N = |S|$. Fiecare punct $p \in S$ este descris prin coordonatele sale carteziene (x, y). Triangularizarea $T(S)$ poate fi cercetată ca un graf planar cu mulțimea de noduri S. Prin urmare, numărul de muchii M în $T(S)$ este proporțional[7] cu N.

O soluție simplă în plan este generarea tuturor muchiilor posibile cu extremități în S, sortarea lor după lungime, apoi adăugarea consecutivă în triangularizare. În proces se verifică, dacă muchia curentă intersectează muchiile adăugate anterior. Dacă intersecțiile lipsesc, muchia este adăugată, în caz contrar, se trece la cercetarea muchiei următoare.

[7] Conform formulei Euler: $M \le 3N - 6$.

Numărul total de muchii posibile pe punctele din S este $N^2/2$. Fiecare muchie poate fi cercetată ca un segment descris prin extremitățile sale:

```
type segment=record
          p1,p2:nod; l:real;
     end;
```

Prin urmare, verificarea intersecției muchiilor poate fi realizată folosind o procedură identică cu procedura de verificare a intersecției segmentelor (§ 2.3).

```
Function intersect (A,B: segment): boolean;
Begin
  If Sarrus(A.p2,A.p1,B.s1)*Sarrus(A.p2,A.p1,B.s2)≤0 and
     Sarrus(B.s2,B.s1,A.p1)*Sarrus(B.s2,B.s1,A.p2)≤0
  then
     Intersect:= true else Intersect:= false
End;
```

Calculul distanței (lungimii muchiei) dintre două puncte $p_i, p_j \in S$ poate fi realizat printr-o funcție elementară:

```
Function distant (p[i],p[j]: nod): real;
Begin
    Distant:=
      sqrt(sqr(p[i].x-p[j].x)+ sqr(p[i].y-p[j].y));
End;
```

Implementarea în pseudocod

Pas 1 `m ← 0`

Pas 2 `Pentru toţi i de la 1 la N`

`Pentru toţi j de la 1+i la N`

a) `m↑`

b) `Muchie[m].l ← distant(p[i],p[j])`

c) `Muchie[m].st ← p[i]`

d) `Muchie[m].fin ← p[j]`

Pas 3 `qsort(muchie,m);`

Pas 4 `k←0`

Pas 5
```
Pentru toţi i de la  1 la  M
   z ← false
   Pentru toţi j de la 1 la  k
   Dacă intersect(Muchie[i],Triang[j]) atunci z←true
    Dacă NOT z atunci
              a)  k↑;
              b)  Triang[k] ← Muchie[i]
```

Numărul total de muchii generate este proporțional cu N^2. Sortarea lor va necesita un număr de operații proporțional cu $N^2 \log(N)$. Complexitatea pasului 5 este determinată de numărul de verificări ale intersecțiilor, care nu depăşeşte N^3. Prin urmare, complexitatea algoritmului greedy este $O(N^3)$, ceea ce lasă de dorit pentru valori mari ale lui N.

4.2. Triangularizarea poligoanelor[8] convexe

Este o problemă elementară, care poate fi rezolvată în timp liniar. Fie P un poligon convex, dat prin lista vârfurilor $p_1, p_2, ..., p_N$ în ordinea parcurgerii lor. Pentru a construi o triangularizare în P, este suficient să fie construite segmen-

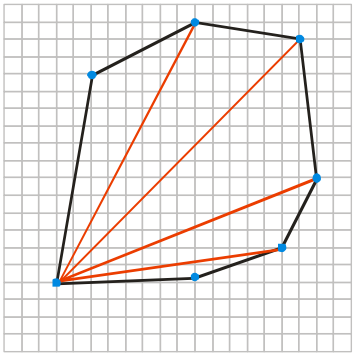


Fig. 4.2. Triangularizarea unui poligon convex

[8] Aici şi în continuare prin poligon se va înțelege conturul acestuia în reuniune cu domeniul lui interior.

tele diagonale $p_1p_3,...,p_1p_{N-1}$ (fig. 4.2). Numărul diagonalelor este proporțional cu N (numărul de vârfuri ale poligonului). Construcția unei diagonale necesită un număr constant de operații. Triangularizarea poligonului convex poate fi utilă pentru calculul ariei lui. Aria poligonului este egală cu suma ariilor triunghiurilor din triangularizare.

4.3. Triangularizarea poligoanelor simple[9].

Metoda greedy.

Realizarea directă a metodei din compartimentul precedent în cazul poligoanelor simple nu este posibilă, deoarece apar două condiții suplimentare care trebuie verificate:

a) aparține oare muchia curentă interiorului sau exteriorului poligonului triangularizat.
b) muchiile care formează frontiera poligonului urmează să fie incluse în triangularizare indiferent de locul ocupat în lista distanțelor.

Verificarea primei condiții este echivalentă cu verificarea apartenenței mijlocului muchiei la domeniul interior al poligonului. Algoritmul cere rezolvă această problemă este prezentat în (§ 5.1).

Problema a doua se rezolvă elementar: prin includerea muchiilor, care formează frontiera P în începutul listei, care descrie triangularizarea.

Fie P un poligon simplu, dat prin lista de vârfuri $p_1, p_2, \dots, p_N$ în ordinea parcurgerii lor. Algoritmul greedy va fi descris de următorul pseudocod:

Pas 1 `m ← 0`
Pas 2 `Pentru toţi i de la 1 la N`

[9] Un poligon este simplu, dacă laturile lui se intersectează doar în vârfuri.

```
Pentru toţi j de la 2+i la N
 a) m↑
 b) Muchie[m].l    ← distant(p[i],p[j])
 c)Muchie[m].st    ← p[i]
 d)Muchie[m].fin   ← p[j]
   End;
```

Pas 3 `qsort(muchie,m);`

Pas 4

```
Pentru toţi i de la 1 la  N-1

    a)  Triang[i].st   ← p[i]
    b)  Triang[i].fin  ← p[i+1]

    Triang[N].st   ← p[N]
    Triang[N].fin  ← p[1]
    k ← N
```

Pas 5

```
Pentru toţi i de la 1 la M
{ z ← false
  For j ← 1 to k do
   If intersect(Muchie[i],Triang[j]) then z ← true
   If NOT z then
          { x,y ← middle(Muchie[i])
            Dacă apart(x,y,P)atunci
                       { k↑;
                         Triang[k] ← Muchie[i]
```

Calculul coordonatelor mijlocului segmentului necesită un număr constant de operații:

```
Procedure middle(a: segment; var x,y:real);
Begin
 x:=(a.st.x+ a.fin.x)/2; y:=(a.st.y+ a.fin.y)/2;
end;
```

Verificarea apartenenței la domeniul interior al poligonului are o complexitate proporțională cu numărul N de laturi în P. Prin urmare, nu influențează complexitatea pasului 5 şi complexitatea totală a algoritmului .

5. Apropiere şi apartenenţă

Capitolul este consacrat analizei principalelor probleme geometrice formulate pe puncte în plan: determinarea celei mai apropiate perechi de puncte şi apartenența punctului la un domeniu. Soluțiile directe ale problemelor de apropiere şi apartenență sunt relativ simple, dar nu şi optime.

5.1. Cea mai apropiată pereche de puncte

Fie în plan o mulțime de puncte $S = \{s_1, ..., s_N\}$. Se cere să se determine o pereche de puncte

$$s_i^*, s_j^*, i \neq j : d(s_i^*, s_j^*) = \min_{\substack{i=1,...,N \\ j=1,...,N \\ i \neq j}} d(s_i, s_j).$$

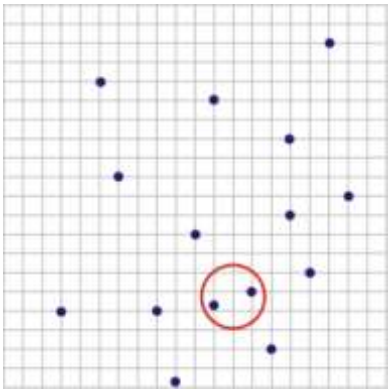


Fig. 5.1 Cea mai apropiată pereche de puncte

Calculul direct al tuturor distanțelor dintre N puncte necesită un număr de operații proporțional cu N^2:

```
index1 ← 1; index2 ← 2;
min ← distance(S1,S2);
    Pentru toţi i de la 1 la  N
      Pentru toţi j de la 1+i la N
            Dacă distance(Si,Sj) < min atunci
                  { min ← distance(Si,Sj)
                  { index1 ← i; index2 ← j;
```

Acelaşi rezultat poate fi obținut într-un timp mai restrâns, folosind algoritmul optim cu o complexitate de $O(N \log N)$

Algoritmul optim

Pentru a determina în timp optim cea mai apropiată pereche de puncte, poate fi folosită tehnologia recursivă *împarte şi stăpâneşte*. Ideea majoră este divizarea la fiecare pas recursiv a mulțimii inițiale în două submulțimi liniar separabile şi rezolvarea problemei pe fiecare submulțime în parte (fig 5.2).

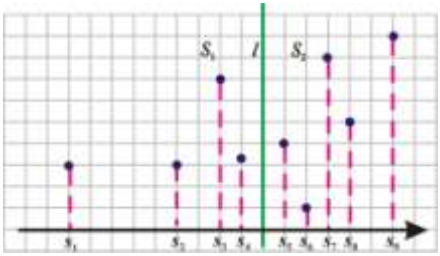


Fig. 5.2. Divizarea mulţimii în submulţimi separabile pentru rezolvarea recursivă a problemei „cea mai apropiată pereche de puncte”

Cazul elementar, care permite calculul direct al soluției, este mulțimea formată din două sau trei puncte. Numărul de operații la acest pas este constant. Specificul problemei este determinarea soluției optime la etapa de asamblare: având două submulțimi S_1 şi S_2, cea mai apropiată pereche de puncte în $S_1 \cup S_2$ poate să fie determinată de o pereche $s', s'' : s' \in S_1, s'' \in S_2$. Se poate demonstra că la etapa asamblării soluției, numărul necesar de verificări la fiecare nivel este liniar față de numărul de puncte în mulțimile asamblate. Numărul de divizări consecutive ale mulțimii în submulțimi „balansate”[10] nu va depăşi $\log(N)$. Dacă numărul de operații necesare pentru determinarea soluției la un nivel de asamblare este proporțional cu N, complexitatea finală a algoritmului va fi $O(N \log N)$. Pentru soluționarea optimă a problemei este necesară o preprocesare a mulțimii: sortarea punctelor după abscisă (se obține şirul sortat X) şi sortarea punctelor după ordonata lor (şirul Y). Având complexitatea

[10] Numărul de elemente în fiecare submulțime diferă cu cel mult o unitate.

$O(N \log N)$, sortarea nu modifică complexitatea finală a algoritmului.

Fie la un nivel de divizare mulțimea S, şirul X sortat după x, şirul Y cu elementele S sortate după y.

Aparent, procesul de asamblare a soluției la un nivel dat are o complexitate $O(N^2)$: maxim N puncte în S_1 şi maxim N puncte în S_2, dintre care urmează să fie calculate distanțele. În realitate, numărul de operații este mult mai mic.

Fie S a fost divizat în submulțimile S_1 şi S_2, pe care au fost determinate distanțele minime δ_1 şi δ_2, precum şi perechile respective de puncte. Se consideră $\delta = \min(\delta_1, \delta_2)$ (fig. 5.3)

Pentru determinarea soluției optime pe $S_1 \cup S_2$ este necesar de a cerceta doar punctele, aflate în fâşia de lățime δ de la dreapta l care separă submulțimile. Celelalte puncte din submulțimi se află, evident, la o distanță mai mare decât δ unul de altul şi nu pot îmbunătăți soluția. Această restrângere nu garantează efectuarea unui număr liniar de operații, deoarece pentru un punct cercetat $s \in S_1$ în fâşia δ poate fi un număr de puncte proporțional cu

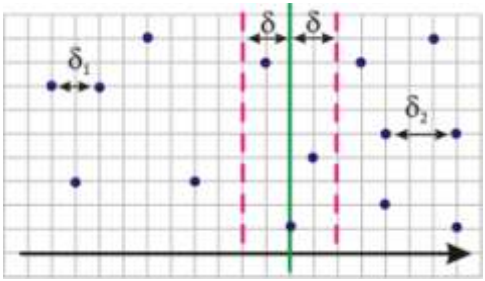


Fig. 5.3. Determinarea punctelor potenţiale pentru soluţia pe $S_1 \cup S_2$

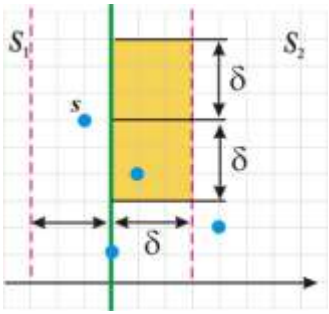


Fig. 5.4. Zona de cercetare în S_2 pentru un punct dat $s \in S_1$

$|S_2|$. O analiză mai detaliată permite excluderea din cercetare a tuturor punctelor care se află în exteriorul dreptunghiului cu latura $\delta \times 2\delta$, asociat punctului s (fig. 5.4). Numărul de puncte din S_2 care se pot afla în această zonă nu poate fi mai mare decât 6, prin urmare numărul de verificări la etapa de asamblare nu va depăşi $6N$.

Folosind şirurile X şi Y, se formează şirul Y', care conţine punctele din S situate în fâşia $[l-\delta, l+\delta]$, sortate după y – mulţimea punctelor care pot îmbunătăţi soluţia. Pentru fiecare element din Y' se calculează distanţele doar până la următoarele 8 elemente: ele reprezintă (în cel mai rău caz) simetric 6 puncte din S_1 plus 6 punc-te din S_2 minus 3 puncte care coincid, minus punctul cercetat (fig. 5.5).

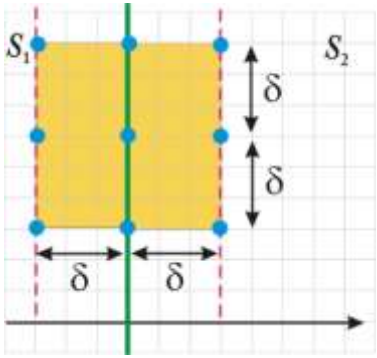


Fig. 5.5. Numărul elementelor consecutive Y' care pot modifica distanţa minimă nu depăşeşte 9

Implementarea în pseudocod

Preprocesare

$X \leftarrow S$, sort(X) {sortare după x}
$Y \leftarrow S$, sort(Y) {sortare după y}

```
Procedure apr2p(S, X, Y)
  Dacă |S| ≥ 4 atunci
        formează S1, S2, X1, X2, Y1, Y2
        δ ← min (apr2p(S1, X1, Y1), apr2p(S2, X2, Y2) )
        formează Y'
        Pentru toţi i de la 1 la |Y'|
          pentru toţi j de la 1 la 8
          dacă distance(Y'[i], Y'[i+j]) < δ atunci
                          δ ← distance (Y'[i], Y'[i+j])
        Return δ
      în caz contrar      Return distanța minimă în S
{calculată direct}
```

5.2. Poligonul şi diagrama Voronoi

O altă problemă de apropiere în plan este problema determinării poligonului Voronoi pentru o mulțime de puncte.

Fie în plan mulțimea de puncte $S = \{p_1, ..., p_N\}$. Fiecare punct p_i, $i = \overline{1, N}$, este descris prin coordonatele sale carteziene $p_i \leftarrow (x_i, y_i)$. Pentru un punct dat $p_i \in S$ se cere să se determine poligonul $V(i)$ care va conține toate punctele planului, mai apropiate de p_i decât de oricare alt punct $p_j \in S$, $p_i \neq p_j$ (fig. 5.6).

Problema generalizată pentru toate punctele mulțimii S este cunoscută sub numele *diagrama Voronoi* (fig 5.7).

De menționat, că pentru unele puncte ale mulțimii S, domeniul determinat de poligonul Voronoi poate fi unul infinit. Se poate demonstra că această proprietate o posedă punctele, care formează înfăşurătoarea convexă a mulțimii S.

Algoritmul direct pentru determinarea poligonului Voronoi $V(i)$ se bazează pe următoarea observație:

Dreapta l perpendiculară pe mijlocul segmentului p_i, p_j conține punctele egal depărtate atât de p_i cât şi de p_j. Punctele mai apropiate de p_i decât de p_j vor forma semiplanul de-

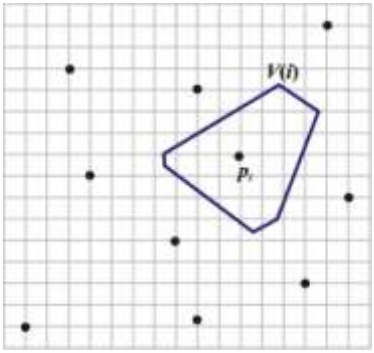


Fig. 5.6. Poligonul Voronoi *V*(i) pentru punctul p_i.

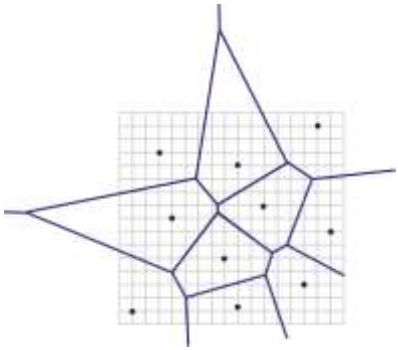


Fig. 5.7 Diagrama Voronoi pentru mulţimea *S*

terminat de dreapta l, care conține şi punctul p_i.

Fiecare punct $p \in S$ este descris prin coordonatele sale carteziene (x, y). Prin urmare, pentru perechea de puncte p_i, p_j,

- mijlocul m al segmentului p_i, p_j va fi determinat de punctul cu coordonatele:
  $x_m = \frac{x_i + x_j}{2}$,
  $y_m = \frac{y_i + y_j}{2}$;
- dreapta d, care conține segmentul cu extremitățile p_i, p_j va fi descrisă de ecuația $Ax + By + C = 0$, unde
  $A = y_j - y_i$
  $B = x_i - x_j$
  $C = x_j y_i - x_i y_j$
- orice dreaptă l', perpendiculară pe segmentul cu extremitățile p_i, p_j va avea ecuația generală de forma $Bx - Ay + C' = 0$;
- pentru dreapta l, perpendiculară pe segmentul cu extremitățile p_i, p_j, şi care trece prin mijlocul acestuia, valoarea coeficientului C este: $Ay_m - Bx_m$

Odată ce este cunoscută ecuația dreptei l perpendiculare pe segmentul p_i, p_j, care trece prin mijlocul acestuia, poate fi determinat semiplanul $R(j)$: $\forall p \in R(j), \quad d(p, p_i) < d(p, p_j)$.[11]

Atunci $V(i) = \bigcap_{\substack{j= \\ j \neq i}}$

[11] Pentru două puncte a,b date prin coordonatele $(x_a, y_a), (x_b, y_b)$
$d(a,b) = \sqrt{|x_a - x_b|^2 + |y_a - y_b|^2}$

Pentru realizarea algoritmului poate fi construit un poligon R de „restrângere" a planului, astfel încât $S \in R$. Cea mai simplă formă a poligonului R este un dreptunghi având laturile paralele cu axele de coordonate, determinat de punctele diagonale $(x_{min}, y_{min}), (x_{max}, y_{max})$:

$$x_{min} = \min_{i=1,\dots,N} (x_i) - 1, \; x_{max} = \max_{i=1,\dots,N} (x_i) + 1,$$

$$y_{min} = \min_{i=1,\dots,N} (y_i) - 1, \; y_{max} = \max_{i=1,\dots,N} (y_i) + 1.$$

Complexitatea algoritmului direct pentru construirea poligonului Voronoi $V(i)$ este $O(N^2)$:

Etapa 1: determinarea poligonului de restrângere necesită un număr de operaţii, proporţional cu N;

Etapa 2: determinarea ecuaţiei dreptei l, perpendiculare pe segmentul p_i, p_j necesită un număr de operaţii mărginit de o constantă.

Etapa 3: determinarea intersecţiei dreptei l cu poligonul R şi restrângerea ulterioară a acestuia necesită un număr de operaţii proporţional cu numărul de laturi ale poligonului R (nu mai mare decât N).

Etapele 2 - 3 se repetă pentru toate punctele $p \in S, \quad p \neq p_i$. Prin urmare, numărul de operaţii necesare pentru determinarea poligonului Voronoi va fi proporţional cu N^2.

Utilizarea algoritmului direct pentru determinarea diagramei Voronoi nu este eficientă: complexitatea algoritmului creşte până la $O(N^3)$.

Realizarea eficientă a algoritmului de determinare a diagramei Voronoi se bazează pe tehnica divizării consecutive a problemei în mulţimi liniar separabile, până la atingerea cazurilor elementare, şi asamblarea ulterioară a soluţiei [11, pag 258 - 269].

Fie $S = S_1 \cup S_2$ şi au fost construite diagramele Voronoi $DV(S_1), DV(S_2)$. (fig. 5.8) Pentru „asamblarea" soluției vor fi realizate următoarele etape:

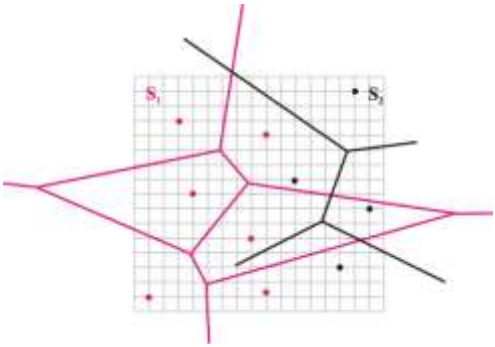


Fig. 5.8. Diagramele Voronoi DV(S_1) şi DV(S_2) pînă la concatenare.

Etapa 1: determinarea vectorilor de intrare (ieşire) a lanțului de concatenare: se construiesc muchiile lipsă (două la număr) pentru înfăşurătoarea convexă comună a $S_1 \cup S_2$, se determină mijloacele acestor muchii şi se trasează vectorii perpendiculari pe acestea, şi care trec prin punctele de mijloc. Pentru muchia superioară vectorul este orientat spre muchie, pentru cea inferioară – de la ea (fig. 5.9). Din înfăşurătoarea convexă pentru $S_1 \cup S_2$ sunt eliminate lanțurile interioare de muchii ale înfăşurătoarelor separate pentru S_1 şi S_2.

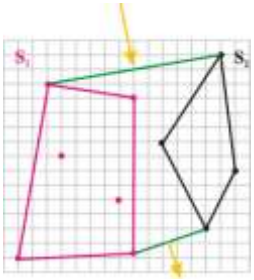


Fig. 5.9 Construirea înfăşurătoarei convexe pentru S prin adăugarea muchiilor. Trasarea vectorilor de intrare (ieşire) a lanțului de concatenare

Etapa 2: Construirea lanțului de concatenare. Construcția lanțului începe pe vectorul superior, dintr-un punct care precede toate intersecțiile vectorului cu razele diagramelor construite anterior. Lanțul se construieşte de-a lungul vectorului, până la intersecția cu una din muchiile diagramelor $DV(S_1)$ sau $DV(S_2)$. În punctul de intersecție direcția vectorului se modifică. (fig. 5.10)

Modificarea direcției lanțului este determinată de modificarea perechii de puncte între care se trasează acesta. Fie,

inițial, lanțul marchează „frontiera” dintre punctele p_i, p_j. Atingerea unei muchii a diagramelor construite anterior semnifică fie înlocuirea p_i prin p_k (dacă muchia atinsă separă p_i, p_k) fie înlocuirea p_j prin p_k (dacă muchia atinsă separă perechea p_j, p_k). Direcția nouă a lanțului de concatenare este determinată de normala mediană la segmentul ce unește perechea nou creată. Procesul ia sfârșit în momentul atingerii vectorului inferior al lanțului de concatenare.

Etapa 3: Modificarea muchiilor diagramei. Muchiile infinite (semidrepte) intersectate de lanțul de concatenare se transformă în segmente delimitate de originea semidreptei şi punctul de intersecție a acesteia cu lanțul de concatenare.

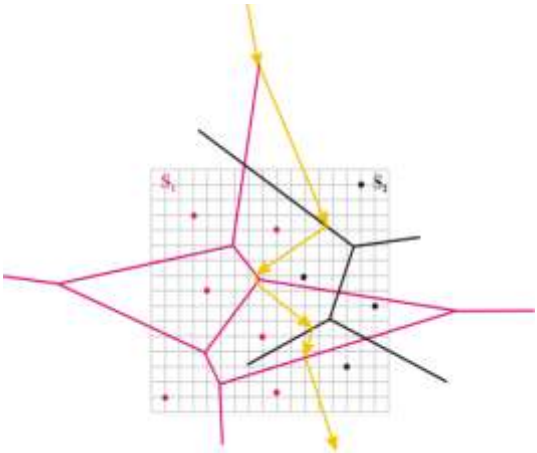


Fig. 5.10. Construirea lanţului de concatenare

Muchiile finite (segmentele) îşi modifică unul din punctele extreme, fiind înlocuit cu punctul de intersecție cu lanțul de concatenare.

Muchiile din $DV(S_1)$, situate integral în dreapta de lanțul de concatenare, şi muchiile din $DV(S_2)$, situate integral în stânga, se exclud din diagrama finală.

Cazul elementar se obține pentru $2 \le |S| \le 3$. Procesarea acestuia este realizată printr-un număr de operații mărginit de o constantă.

Divizarea consecutivă a mulțimii pentru obținerea cazurilor elementare necesită o sortare prealabilă a punctelor din S după abscisa lor. Complexitatea preprocesării este $O(N \log N)$. Numărul de divizări consecutive ale mulțimii curente în două submulțimi „aproape egale”[12] este proporțional cu $\log(N)$. Pentru concatenarea soluțiilor parțiale, în cazul alegerii structurilor de date adecvate este necesar un număr de operații proporțional cu numărul total de muchii din soluțiile parțiale. Deoarece diagrama Voronoi este o divizare planară a unei mulțimi din N puncte, numărul de muchii va fi proporțional cu N. Prin urmare, complexitatea totală a algoritmului optim este $O(N \log N)$.

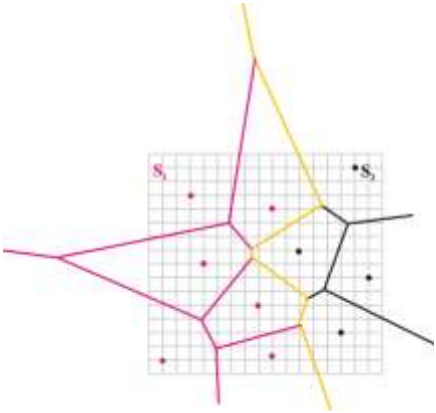


Fig. 5.11. Concatenarea diagramelor

5.3. Apartenența punctului la un domeniu

Apartenența punctului la un poligon convex

Problema determinării apartenenței unui punct la un poligon convex este una simplă şi poate fi rezolvată cu ajutorul algoritmilor cercetați anterior. Se observă uşor (fig. 5.12) că poziția unui punct interior față de fiecare din vectorii determinați de vârfurile consecutive ale poligonului este una şi aceeaşi – la dreapta, dacă vârfurile sunt parcurse în direcția mişcării acelor

[12] S_1 ”aproape egală” cu S_2 dacă $\left||S_1|-|S_2|\right| \le 1$

de ceasornic, şi la stânga – în cazul parcurgerii vârfurilor în direcţie opusă.

Poziţia punctului *s* faţă de un vector $\overrightarrow{p_i p_j}$ poate fi determinată în timp constant (§ 3.1). Prin urmare, complexitatea algoritmului este proporţională doar cu numărul de laturi ale poligonului. Fie punctual *s* de coordonate (x_s, y_s) şi poligonul convex $P = (p_1, p_2, \ldots, p_N)$ cu N laturi, descris prin lista vârfurilor, parcurse consecutiv (coordonatele vârfurilor sunt stocate în tabloul liniar de articole **P** cu câmpurile **x** şi **y**). Pentru a simplifica implementarea, în lista de vârfuri ale poligonului este inclus un vârf virtual $p_{N+1} \leftarrow p_1$.

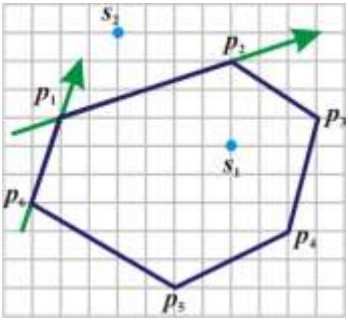


Fig. 5.12. Punctele interioare sunt plasate de aceeaşi parte a vectorilor determinaţi de vârfurile consecutive ale poligonului, poziţia celor exterioare poate varia.

```
function apart : boolean;
var i : integer;

function verific_punct(x1,y1,x2,y2,x3,y3:real):boolean;
  begin
   if  x1*y2+x2*y3+y1*x3-x3*y2-x2*y1-y3*x1 > 0 then
       verific_punct:=true else verific_punct:=false;
  end;

begin
  apart:=true;
  for i:=1 to N do
  if not
  verific_punct(p[i].x,p[i].y,p[i+1].x,p[i+1].y,s.x,s.y)
  then apart:=false;
end;
```

Apartenența punctului la un poligon stelat

Fie un poligon arbitrar $P=(p_1,p_2,...,p_N)$ cu N laturi. Dacă în P există un punct interior $\boldsymbol{c}$, astfel încât toate segmentele $[c,p_1],[c,p_2],...,[c,p_N]$ aparțin integral P, poligonul se numeşte *stelat*. (fig. 5.13)

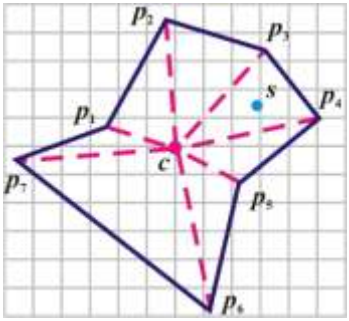


Fig. 5.13. P - poligon stelat

În cazul, în care punctul interior *c* este cunoscut apriori, determinarea apartenenței unui punct arbitrar *s* la domeniul interior al poligonului se reduce la verificarea existenței unui triunghi[13] Δ , $i=\overline{1,N}$ care să conțină punctul s în calitate de punct interior.

Numărul triunghiurilor este *N*, numărul de operații necesare pentru verificarea apartenenței punctului la interiorul unui triunghi este mărginit de o constantă. Prin urmare complexitatea determinării apartenenței punctului la un poligon stelat va fi $O(N)$ în condiția că punctul *c* este cunoscut.

Apartenența punctului la un poligon simplu

Problema apartenenței unui punct *s* la domeniul determinat de un poligon simplu *P* poate fi rezolvată prin triangularizarea *P* şi prin verificarea ulterioară a apartenenței *s* la unul din triunghiurile formate în procesul triangularizării.

[13] Vîrful p_{N+1} este unul virtual şi coincide cu p_1

Un algoritm mai eficient este bazat pe numărarea intersecțiilor ale dreptei orizontale l care trece prin punctul dat s cu laturile poligonului P. (fig. 5.14). Se va cerceta partea dreptei spre stânga de s. Pentru latura curentă se determină :

- poziția punctului față de aceasta
- intersecția laturii cu semidreapta l.

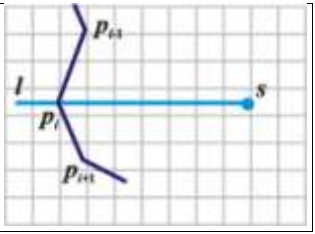
Fig. 5.14. Numărul de intersecţii ale semidreptei l cu laturile poligonului determină apartenenţa punctului s la interiorul poligonului P.

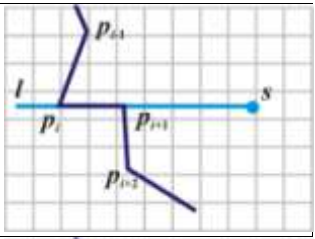
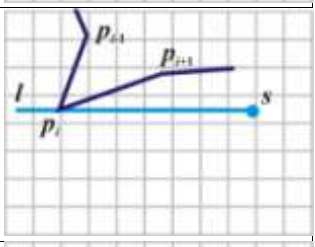
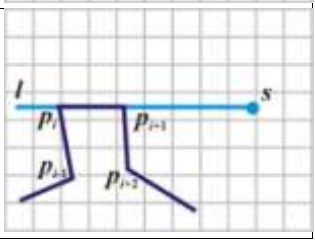
Dacă numărul intersecțiilor spre stânga de s este impar – punctul se află în interiorul poligonului; pentru un număr par de intersecții, punctul se află în exterior. Demonstrația este elementară: la parcurgerea spre stânga, prima intersecție marchează intrarea semidreptei în interiorul poligonului, următoarea – ieşirea. Fiecare pereche următoare de intersecții are aceeaşi semnificație.

Pentru stabilirea intersecției semidreptei cu latura curentă pot fi utilizate metodele descrise în § 2.2.

Cazuri speciale.

dreapta l trece printr-un vârf p_i al poligonului P la stânga de s. Vârfurile p_{i-1} şi p_{i+1} se află de părți diferite ale dreptei l. Numărul de intersecții se incrementează cu 1.



dreapta l conține latura $p_i\,p_{i+1}$ a poligonului P la stânga de s. Vârfurile p_{i-1} şi p_{i+2} se află de părți diferite ale dreptei l. Numărul de intersecții se incrementează cu 1.	
dreapta l trece prin un vârf p_i al poligonului P la stânga de s. Vârfurile p_{i-1} şi p_{i+1} se află de aceeaşi parte a dreptei l. Numărul de intersecții nu se incrementează.	
dreapta l conține latura $p_i\,p_{i+1}$ a poligonului P la stânga de s. Vârfurile p_{i-1} şi p_{i+2} se află de aceeaşi parte a dreptei l. Numărul de intersecții nu se incrementează.	

Numărul de laturi procesate este N. Determinarea intersecției laturii cu dreapta l este realizată într-un număr constant de operații. Procesarea cazurilor speciale este realizată de asemenea într-un număr de operații mărginit de o constantă. Prin urmare, complexitatea algoritmului este $O(N)$.

5.4. Nuclee

Printre problemele geometrice de calcul se numără şi problema *nucleului* poligonului simplu. Nucleul unui poligon simplu P este, în general, format din mulțimea de puncte $Q \subseteq P$, astfel încît:

$$\forall x \in Q, \forall y \in P, [x, y] \in P$$

Cu alte cuvinte, nucleul poligonului este mulțimea de puncte ale poligonului, care, fiind unite cu orice alt punct al poligonului, formează un segment ce aparține interiorului poligonului (fig. 5.15a).

Nu întotdeauna un poligon simplu are nucleu nevid (fig. 5.15.b).

Nucleul poligonului (dacă el există) este şi el un poligon, dar, spre deosebire de poligonul inițial, este întotdeauna convex.

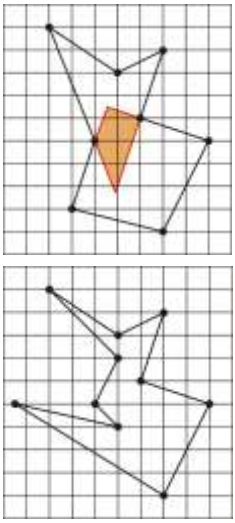


Fig. 5.15. a) Poligon simplu cu nucleu, **b**) poligon simplu fără nucleu

Algoritmul pentru determinarea nucleului unui poligon simplu

Date inițiale. Fie un poligon simplu P cu `N` vârfuri. Se consideră că poligonul este descris prin lista vârfurilor sale $(p_1, p_2, ..., p_{N-1}, p_N)$, parcurse în direcția mişcării acelor de ceasornic. Fiecare vârf p_i este descris de coordonatele sale (x_i, y_i).

Algoritmul de determinare a nucleului necesită o construcție secundară: un poligon convex Q care, inițial, conține poligonul P. Pentru determinarea poligonului Q pot fi abordate două metode:

a) construirea înfăşurătorii convexe pentru P;
b) construirea unui dreptunghi care conține poligonul P în interiorul său.

Metoda a doua este mult mai simplă. Ea se reduce la determinarea, pentru coordonatele vârfurilor poligonului, a valorilor minime şi maxime ale abscisei şi ale ordonatei. În calitate de Q poate fi considerat dreptunghiul cu vârfurile $(x_{\max}+1, y_{\max}+1)$, $(x_{\max}+1, y_{\min}-1)$, $(x_{\min}-1, y_{\min}-1)$, $(x_{\min}-1, y_{\max}+1)$. Extinderea valorilor extreme nu este o condiție obligatorie.

După construcția poligonului Q poate fi realizată nemijlocit determinarea nucleului. Metoda (sau cel puțin descrierea ei în limbajul uman) este foarte simplă:

Se analizează consecutiv laturile poligonului P, fiind parcurse în direcția mişcării acelor de ceasornic. Latura curentă (p_i, p_{i+1}) se cercetează ca o dreaptă l. Se determină partea poligonului Q, care se află în stânga vectorului $\overrightarrow{p_i p_{i+1}}$ şi se exclude din Q. Dacă poligonul Q se este situat integral în stânga de vectorul $\overrightarrow{p_i p_{i+1}}$, cercetarea ia sfârşit – poligonul P nu are nucleu.

Procedeul se repetă până nu sunt analizate toate laturile poligonului P. În final, Q conține nucleul lui P. (fig. 5.16).

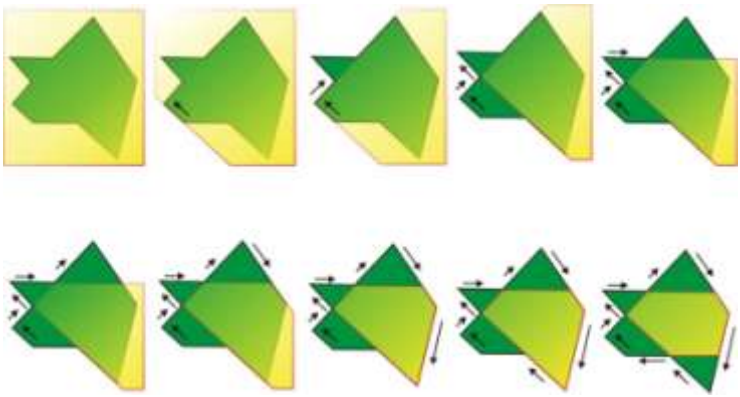


Fig. 5.16. Construirea consecutivă a nucleului poligonului simplu

Implementarea în pseudocod

Pas 1. Inițializare

La sfârşitul listei de vârfuri (după p_N) se adaugă vârful virtual $p_{N+1} = p_1$. Se construieşte poligonul Q (nucleul inițial), conform uneia din metodele descrise anterior. $i \leftarrow 1$;

Pas 2. Procesarea laturii *i*.

Fie $Q = (q_1, q_2, ...q_k)$. Pentru dreapta l ce conține latura i a poligonului P definită de vârfurile (p_i, p_{i+1}) se determină punctele de intersecție cu poligonul Q, precum şi laturile intersectate. Fie d_1, d_2 punctele de intersecție a dreptei l cu Q, iar laturile intersectate: (q_j, q_{j+1}) şi (q_k, q_{k+1})[14].

a. Se formează poligoanele $Q_1 = (d_1, q_{j+1}, \dots, q_k, d_2)$ şi $Q_2 = (d_2, q_{k+1}, \dots, q_N, q_1, \dots, q_j, d_1)$

b. Se determină $Q^* \in \{Q_1, Q_2\}$ care se află în dreapta vectorului $\overrightarrow{(p_i p_{i+1})}$.

c. $Q \leftarrow Q^*$

Pas 3. Repetare

Dacă $i < N$ **atunci**

{ $i \uparrow$

revenire la pas 2

în caz contrar STOP - Q este nucleul.

Complexitatea algoritmului este de $O(N^2)$. Se prelucrează **N** laturi ale poligonului P. Pentru fiecare latură se verifică intersecția cu maximum **N** laturi ale poligonului Q. Fiecare intersecție este determinată într-un număr de operații mărginit de o constantă.

---

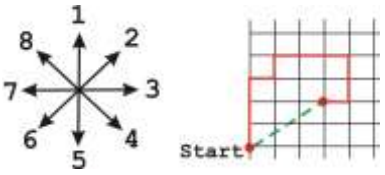
[14] Dacă l nu intersectează poligonul Q, se verifică doar poziția Q față de $\overrightarrow{p_i p_{i+1}}$. Dacă Q e poziționat în stânga, nucleul final este vid, altfel Q nu se modifică.

6. Probleme geometrice de concurs

6.1 Robot

Un robot punctiform poate executa instrucțiuni de deplasare în 8 direcții (1 – nord, 2 – nord-est, 3 – est, 4 – sud-est, 5 – sud, 6 – sud-vest, 7 – vest, 8 – nord-vest). Lungimea pasului robotului este 1 pentru direcțiile 1, 3, 5, 7 şi $\sqrt{2}$ pentru direcțiile 2, 4, 6, 8. Numărul de paşi efectuați în direcția aleasă este un parametru al instrucțiunii de deplasare.

Fiind dat un set de instrucțiuni, să se determine coordonatele punctului final în care se va deplasa robotul.



Astfel, pentru setul (1, 3) (3, 1) (1, 1) (3, 3) (5, 2) (7, 1) coordonatele finale vor fi (3, 2) .

Cerinţă

Să se scrie un program care, după un set de instrucțiuni, să determine coordonatele punctului final în care se va deplasa robotul. Se consideră că axa *Ox* e orientată spre est, iar *Oy* – spre nord. Inițial robotul se află în originea sistemului de coordonate (0, 0).

Date de intrare

Prima linie a fişierului de intrare conține numărul **N** – numărul de instrucțiuni (**1≤N≤40**). Următoarele **N** linii conțin instrucțiunile propriu-zise – numărul direcției (un număr întreg de la `1` la `8`) şi numărul de paşi (un număr întreg de la `1` la `1000`), separate prin spațiu.

Date de ieşire

Unica linie a fişierul de ieşire va conține două numere întregi **X** şi **Y** separate prin spațiu – coordonatele punctului final în care se deplasează robotul.

Exemple

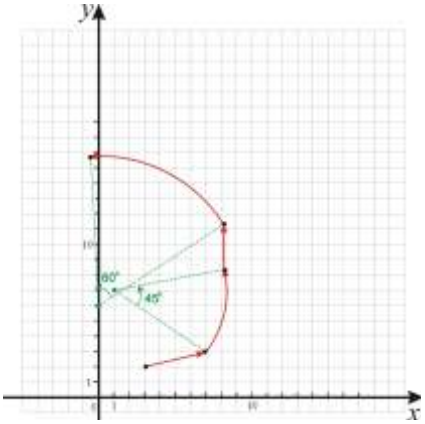
date.in	date.out
6 1 3 3 1 1 1 3 3 5 2 7 1	3 2
1 8 10	-10 10

Rezolvare

```
program p01;
var   I,N:integer;
      D,L,X,Y:Longint;
begin
  assign(Input,'date.in'); reset(Input);
  read(N);
  X:=0; Y:=0;       {fixarea pozitiei initiale}
  for I:=1 to N do
   begin             {modelarea deplasarii}
    read(D,L);
    case D of
     1: Y:=Y+L;                  { deplasare nord cu L}
     2: begin X:=X+L; Y:=Y+L; end; {nord-est cu L}
     3: X:=X+L;                    {est cu L}
     4: begin X:=X+L; Y:=Y-L; end; {sud-est cu L}
     5: Y:=Y-L;                    {sud cu L}
     6: begin X:=X-L; Y:=Y-L; end; {sud-vest cu L}
     7: X:=X-L;                    {vest cu L}
     8: begin X:=X-L; Y:=Y+L; end; {nord-vest cu L}
  end;
 end;
  assign(Output,'date.out'); rewrite(Output);
  write(X,' ',Y);
  close(Input); close(Output);
end.
```

6.2.Robot II

Sistemul de comandă al unui robot care poate executa instrucțiuni de deplasare s-a deteriorat. Robotul continuă să primească instrucțiuni, dar le interpretează nu întotdeauna corect. Fiecare instrucțiune este un triplet de numere (**u,vx,vy**) întregi, poziti-ve. Dacă **u** > 90, atunci robotul se deplasează cu **vx** după axa *Ox* şi cu **vy** după axa *Oy*. Dacă, însă **u** ≤ 90, robotul se deplasează pe un arc de măsura **u** a cercului de (**vx,vy**) împotriva mişcării acelor de ceasornic.



Raza cercului este segmentul care uneşte robotul cu centrul cercului.

Cerinţă

Să se scrie un program care, după coordonatele inițiale ale robotului şi un set dat de instrucțiuni, determină punctul final în care va fi poziționat acesta.

Date de intrare

Prima linie a fişierului de intrare conține două numere întregi – coordonatele inițiale ale robotului. Următoarele linii conțin câte o instrucțiune, formată din trei numere întregi, separate prin spațiu.

Date de ieşire

Unica linie a fişierul de ieşire va conține două numere `X` şi `Y`, separate prin spațiu – coordonatele finale ale robotului, indicate cu cel puțin 3 cifre după virgulă.

Exemplu

date.in	date.out
3 2 130 4 1 45 1 7 91 0 3 60 0 6	-0.653 15.697

Rezolvare

```
program p02;
const pi=3.141592;
type point=record
           x,y : real;
     end;
 var P: point;
     alfa,xn,yn:real;
procedure move(var P:point; vx,vy:real);
 begin
  P.x:=P.x+vx;
  P.y:=P.y+vy;
 end;
procedure rotate (var P:point; u,vx,vy:real);
 var old:point;
 begin
  old:=P;
  P.x:=vx+(old.x-vx)*cos(u*pi/180)-
                 (old.y-vy)*sin(u*pi/180);
  P.y:=vy +(old.x-vx)*sin(u*pi/180)+
                 (old.y-vy)*cos(u*pi/180);
 end;
begin
  assign(input,'data.in'); reset(input);
  assign(output,'data.out'); rewrite(output);
  readln(P.x,P.y);
  while not eof do
    begin
     readln(alfa,xn,yn);
     if alfa>90 then move(P,xn,yn)
                else rotate(P,alfa,xn,yn);
    end;
  writeln(P.x:10:3,' ',P.y:10:3);
  close(input); close(output);
 end.
```

6.3. Piatra

... şi unde nu porneşte stânca la vale, săltând tot mai sus de un stat de om; şi trece prin gardul şi prin tinda Irinucăi, pe la capre, şi se duce drept în Bistriţa, de clocotea apa!

Ion Creangă. Amintiri din copilărie

Piatra pornită de Ion Creangă se rostogolea în linie dreaptă, dărâmând totul în calea sa. Casa Irinucăi are mulți pereți, unii nimerind în calea pietrei. Fiind date coordonatele a două puncte prin care trece piatra şi extremitățile segmentelor care descriu baza pereților casei, să se determine, câți pereți vor fi dărâmați de piatra în cădere. Peretele atins într-un punct extrem rămâne întreg.

Cerinţă

Scrieți un program care determină numărul de pereți dărâmați de piatră.

Date de intrare

Fişierul de intrare conține **`N (N < 1000)`** linii a câte patru numere întregi. Prima linie conține descrierea a două puncte prin care trece piatra, în forma $x_1, y_1, x_2, y_2 \quad |x_i, y_i| < 500$. Următoarele **`N-1`** linii conțin descrierea coordonatelor extremităților bazelor pereților, în aceeaşi consecutivitate, cu aceleaşi restricții de valori.

Date de ieşire

Unica linie a fişierul de ieşire va conține un număr întreg – numărul de pereți dărâmați de piatră.

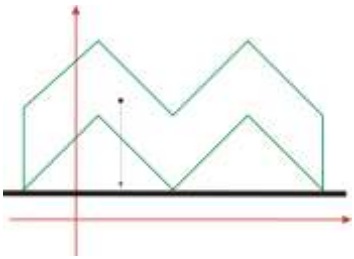
Exemplu

date.in	date.out
2 1 9 14 2 4 1 8 1 12 3 9 4 14 6 11 7 14 9 10 9 14 7 16 6 10 8 8 3 6 6 6 4 2 4 5 11 10 12 14 8 6 10 8 12 8 14 4	4

Rezolvare

```
program p03;
type point=record      x,y: real;      end;
     segment=record    e1,e2: point;   end;
var g: array[1..1000] of segment;
    l: segment;
    n,i,k: integer;

function sarrus(p1,p2,p3:point): real;
 begin    sarrus:=p1.x*p2.y+p2.x*p3.y+p1.y*p3.x
            -p3.x*p2.y-p3.y*p1.x-p1.y*p2.x;
 end;

begin
 assign(input, 'date.in'); reset(input); n:=0;
 readln(l.e1.x,l.e1.y,l.e2.x,l.e2.y);
 while not eof do
    begin
      inc(n); readln( g[n].e1.x,
      g[n].e1.y,g[n].e2.x,g[n].e2.y);
    end;
 k:=0;
 for i:=1 to n do
 if
 sarrus(l.e1,l.e2,g[i].e1)*sarrus(l.e1,l.e2,g[i].e2) < 0
 then inc(k);
 assign(output, 'date.out'); rewrite(output);
 writeln(k); close(input); close(output);
end.
```

6.4 Carcase



Fie o carcasă, având forma unui poligon simplu. Pentru a fi poziționată vertical pe o suprafață plană, carcasa trebuie să aibă centrul de masă situat strict între 2 puncte de contact cu suprafața. Centrul de masă este întotdeauna un punct interior al carcasei şi nu coincide cu nici un vârf al ei.

Cerinţă

Să se scrie un program care va determina numărul pozițiilor în care poate fi stabilizată vertical carcasa

Date de intrare

Prima linie a fişierului de intrare conține trei numere întregi, separate prin spațiu: N ($3 \le N \le 1000$) - numărul de vârfuri ale carcasei şi x_c, y_c ($-1000 \le x_c, y_c \le 1000$) – coordonatele centrului de masă.

Urmează N linii ce conțin câte 2 numere întregi x_i, y_i ($-1000 \le x_i, y_i \le 1000$), separate prin spațiu, – coordonatele vârfurilor poligonului în ordinea parcurgerii lor.

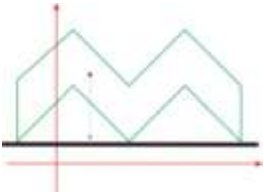
Date de ieşire

Fişierul de ieşire va conține un număr întreg – numărul de poziții în care poate fi stabilizată vertical carcasa.

Exemplu

drag.in	drag.out
10 6 16 3 14 13 4 23 14 33 4 33 14 23 24 13 14 3 24 -7 14 -7 4	3

Rezolvare



Deoarece carcasa se sprijină pe careva puncte extreme ale poligonului, problema poate fi divizată în două subprobleme relativ simple:

1) determinarea înfăşurătoarei convexe a vârfurilor poligonului;
2) verificarea dacă un triunghi conține un unghi obtuz.

Prima subproblemă este rezolvată cu ajutorul algoritmului descris anterior (§ 3.2).

După determinarea înfăşurătoarei convexe, problema poate fi reformulată în felul următor:

Fie un poligon convex P şi un punct interior c, unit prin linii drepte cu vârfurile poligonului. În câte dintre triunghiurile formate înălțimea construită din punctul c se proiectează într-un punct interior al laturii poligonului care aparține triunghiului?

Evident, înălțimea construită din **c** se proiectează pe latură dacă nici unul dintre unghiurile alăturate laturii poligonului nu este obtuz. Verificarea unghiurilor se realizează elementar cu ajutorul teoremei cosinusurilor. Complexitatea etapei este liniară după numărul de vârfuri.

6.5. Turnuri

Bitlanda este o țară care se extinde permanent. Pentru a apăra frontierele sale, după fiecare extindere, în noile puncte extreme ale frontierei se construiesc turnuri de veghe. Există N turnuri de veghe, date prin coordonatele lor (x_i, y_i) – numere întregi. Regele Bitlandei, Bytezar, a decis sa trimită la vatră garnizoanele turnurilor care nu se mai află la hotarele țării.

Cerinţă

Scrieți un program, care va determina câte turnuri vor fi lipsite de garnizoane.

Date de intrare

Prima linie a fişierului de intrare conține un număr întreg: N $(1 \le N \le 10000)$ – numărul de turnuri de veghe în Bitlanda.

Urmează N linii ce conțin câte două numere întregi x_i,y_i ($-1000 \le x_i, y_i \le 1000$), separate prin spațiu – coordonatele turnurilor de veghe.

Date de ieşire

Fişierul de ieşire va conține un număr întreg – numărul de turnuri care pot fi lipsite de garnizoane.

Exemplu

turn.in	Turn.out
10 2 1 3 4 6 3 6 6 8 5 10 3 11 7 12 6 9 11 15 8	5

6.6 Atac

Agenția Spațială a planetei Bitterra (ASB) a recepționat un roi meteoritic, care se apropie de planetă. Fiecare stat de pe Bitterra a fost anunțat despre pericol şi a primit lista punctelor de cădere a meteoriților, calculate de ASB. Serviciile pentru situații excepționale ale fiecărui stat urmează să determine, câți meteoriți vor cădea pe teritoriul țării. Frontiera oricărui stat de pe Bitterra este un poligon convex; meteoriții sunt punctiformi. Punctele de pe frontieră nu se consideră aparținând statului. Frontiera poate conține mai multe vârfuri consecutive coliniare.

Cerinţă

Să se scrie un program care va determina numărul de meteoriți, ce cad pe teritoriul unui stat dat.

Date de intrare

Prima linie a fişierului de intrare **`atac.in`** conține numărul **`n`** – numărul de vârfuri ale poligonului, care descrie frontiera unui stat. Următoarele **`n`** linii conțin descrierea consecutivă a vârfurilor frontierei: fiecare linie va conține câte o pereche de numere separate prin spațiu – coordonatele unui vârf. Urmează o linie care conține un număr întreg **`m`** – numărul de meteoriți, apoi **`M`** linii, care conțin câte două numere, separate prin spațiu, – coordonatele punctelor de cădere a meteoriților.

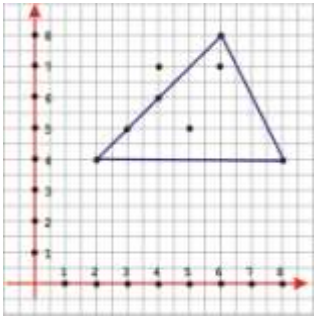
Date de ieşire

Fişierul de ieşire **`atac.out`** va conține un singur număr întreg – cel al meteoriților care cad pe teritoriul statului dat.

Restricţii

1. Coordonatele vârfurilor poligonului şi a punctelor de cădere a meteoriților sunt numere întregi din intervalul **`[-1000000, 1000000]`**
2. **`3 ≤ n ≤ 20000`**
3. **`2 ≤ m ≤ 20000`**

Exemplu

atac.in	atac.out	Explicaţie
4 2 4 8 4 6 8 4 6 4 3 5 4 7 5 5 6 7	2	

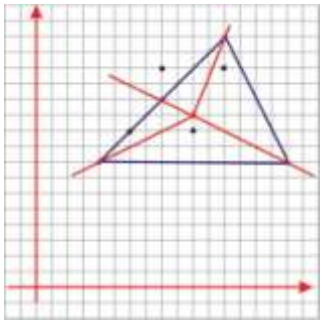
Soluţie:

Formularea abstractă a problemei este următoarea:

Fie în plan un poligon convex P cu **n** vârfuri. În acelaşi plan este dată o mulţime din **m** puncte M. Se cere să se determine câte puncte din M aparţin interiorului P.

Rezolvarea se divide în câteva etape.

1. Determinarea unui punct intern al poligonului (centrul de masă).



Fiind date coordonatele (x_i, y_i), $i = \overline{1,n}$ ale vârfurilor poligonului P, coordonatele centrului de masă pot fi calculate după formula:

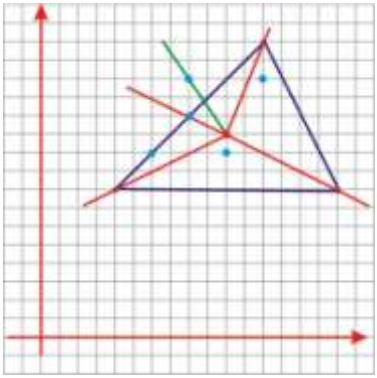
$$x_{cm} = \frac{\sum_{i=1}^{n} x_i}{n}, \quad y_{cm} = \frac{\sum_{i=1}^{n} y_i}{n}.$$

2. Fiind dat un punct intern (**xcm, ycm**) al poligonului, se calculează unghiurile, pe care le formează cu axa Ox semidreptele duse din acesta prin vârfurile (**cx, cy**).

```
function angle(cx,cy:real): real;
```

```
begin
sinus:=(cy -ycm)/ sqrt(sqr(cy-ycm)+sqr(cx-xcm));
cosinus:= (cx -xcm)/ sqrt(sqr(cy-ycm)+sqr(cx-xcm));
if cosinus=0 then
   if sinus>0 then angle:= pi/2
              else if sinus<0 then angle:=3*pi/2
else  begin
      if (sinus>0) and (cosinus>0)
          then angle:=arctan(sinus/cosinus);
      if (sinus>0) and (cosinus<0)
          then angle:=pi-arctan(abs(sinus/cosinus));
      if (sinus<=0)and (cosinus<0)
          then angle:=pi+arctan(abs(sinus/cosinus));
      if (sinus<=0) and (cosinus>0)
          then angle:=2*pi-arctan(abs(sinus/cosinus));
      end;
 end;
```

3. Pentru un punct dat din M se calculează unghiul format de semidreapta determinnată de acesta şi centrul de masă al poligonului cu axa *Ox*. Se foloseşte aceeaşi funcţie de la etapa 2.



```
alfa:=angle(b[i].x,b[i].y)
```

4. Se determină muchia poligonului, intersectată de semidreapta determinată la pasul 3. Pentru optimizarea acestui pas se foloseşte căutarea binară.

```
k:=1; {determinarea vârfului cu unghi maxim}
  while (a[k].u<=a[k+1].u) and (k<n) do k:=k+1;
      {divizarea binara}
      if (alfa < a[k+1].u) or (alfa>a[k].u)
```

```
then begin st:=k; dr:=k+1; end
else begin
      if (alfa < a[1].u)
       then  begin  st:=k+1; dr:=n+1; end
       else  begin  st:=1; dr:=k;     end;
      repeat
       mj:=(st + dr) div 2;
       if a[mj].u >alfa then dr:=mj else st:=mj
      until dr=st+1;
     end;
```

5. Se determină poziția centrului de masă şi a punctului curent din M față de muchia determinată. Dacă sunt de aceeaşi parte, punctul curent este în interior. Pentru determinarea poziției se foloseşte semnul determinantului $\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$ unde $(x_1, y_1), (x_2, y_2)$ sunt coordonatele vârfurilor care determină muchia, iar (x_3, y_3) - coordonatele punctului cercetat (respectiv coordonatele centrului de masă).

```
q:=semn(a[st].x,a[st].y,a[dr].x,a[dr].y,b[i].x,b[i].y);
p:=semn(a[st].x,a[st].y,a[dr].x,a[dr].y,xcm,ycm);
if p*q>0 then inc(s);
```

6.7. Evadare

Un grup de pinguini a decis să evadeze din grădina zoologică. În acest scop ei au săpat un tunel liniar. Din nefericire, zidul grădinii zoologice formează un poligon simplu cu *N* ($3 \leq N \leq 10000$) laturi, astfel încât, ieşind din tunel, pinguinii nu pot afirma cu certitudine dacă se află în interiorul grădinii zoologice sau în afara ei.

Cerinţă

Să se scrie un program care, după coordonatele punctelor extreme ale tunelului şi coordonatele vârfurilor poligonului ce stabileşte perimetrul grădinii zoologice, va determina dacă evadarea s-a soldat cu succes sau nu.

Date de intrare

Prima linie a fişierului de intrare **`evadare.in`** conţine patru numere întregi, separate prin spaţiu, coordonatele punctelor extreme ale tunelului. Următoarele linii conţin descrierea consecutivă a vârfurilor zidului: fiecare linie va conţine câte o pereche de numere, separate prin spaţiu, – coordonatele unui vârf.

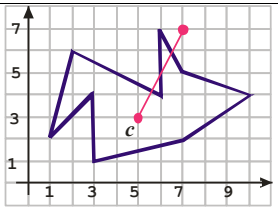
Date de ieşire

Fişierul de ieşire **`evadare.out`** va conţine un singur cuvânt DA – în cazul în care punctul final al tunelului este în afara grădinii zoologice; NU – în caz contrar.

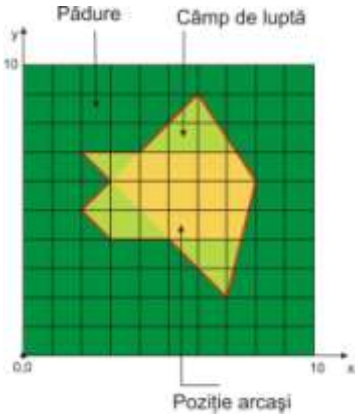
Restricţii

Coordonatele vârfurilor poligonului şi ale punctelor extreme ale tunelului sunt numere întregi din intervalul **`[-1000, 1000]`**

Exemplu

evadare.in	evadare.out	Explicaţie
5 3 7 7 1 2 2 6 6 4 6 7 7 5 10 4 7 2 3 1	DA	

6.8. Arcaşi



Secretul victoriilor faimosului comandant de oşti MegaFlop este strategia lui de alegere a poziției arcaşilor pe câmpul de luptă. Câmpul are forma unui poligon simplu şi e înconjurat de păduri. MegaFlop plasează arcaşii doar pe poziții din care este **văzut** tot câmpul de luptă. Se consideră că arcaşii văd tot câmpul, dacă din orice punct care aparține poziției lor de tragere se poate trage cu săgeata în orice alt punct al câmpului. Traiectoria săgeții este liniară. Nimerind în pădure, săgeata se pierde. Pentru tragere, fiecare arcaş are nevoie de o unitate de suprafață. Astfel, numărul maxim de arcaşi, care pot fi plasați pe poziții este determinat de aria poligonului din care este văzută toată câmpia.

Cerinţă

Să se scrie un program care determină numărul maxim de arcaşi care pot fi plasați pe poziții de tragere în câmpul de luptă.

Date de intrare

Fişierul de intrare **arcas.in** va conține pe prima linie un număr întreg **N** – numărul de vârfuri ale poligonului simplu, care descrie perimetrul câmpului de luptă. Urmează **N** linii care conțin coordonatele vârfurilor poligonului parcurse în sensul acelor de ceasornic, câte un vârf pe linie. Linia **i+1** conține două numere întregi x_i, y_i, separate prin spațiu – coordonatele vârfului **i**.

Date de ieşire

Fişierul de ieşire **arcas.out** va conține un singur număr întreg: numărul maxim de arcaşi, care pot fi plasați pe poziții.

Restricţii

$3 \le N \le 1000$
$0 < x_i, y_i \le 10000$

Exemplu

arcas.in	arcas.out
9 2 5 3 6 2 7 4 7 6 9 8 6 7 2 5 4 3 4	11

Rezolvare

```
program p68;
 type
   lat=record x,y:real; end;
   pol=array[0..1001] of lat;
 var
     nuc,camp:pol;
     i,nnuc,ncamp:integer;
     xint,yint:real;
```

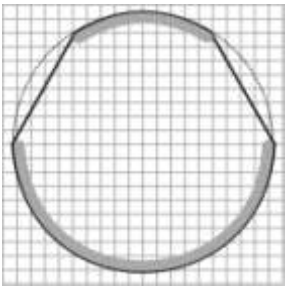
```
procedure init;
  var square : array[1..5,1..2] of integer;
      i: integer;
  begin
{initializare nucleu}
  nuc[1].x:=0;  nuc[1].y:=0;
  nuc[2].x:=0;  nuc[2].y:=10001;
  nuc[3].x:=10001; nuc[3].y:=10001;
  nuc[4].x:=10001; nuc[4].y:= 0;
  nuc[5].x:=nuc[1].x; nuc[5].y:= nuc[1].y;
  nnuc:=4;

{… si initializare poligon}
  readln(ncamp);
  for i:=1 to ncamp do readln(camp[i].x,camp[i].y);
  camp[ncamp+1].x:=camp[1].x;camp[ncamp+1].y:=camp[1].y;
 end;

function intersect
         (al,bl,cl,ad,bd,cd:real;i,j:integer): boolean;

{determină intersecţia dreptei şi laturii
 + coordonate punctului de intersecţie}
begin

{1. dreapta şi latura sunt paralele}
  if Ad*Bl=Bd*Al then begin intersect:=false; exit; end;

{2. dreapta intersectează 2 laturi adiacente în punct
 extrem}
  if (camp[j+1].x=nuc[i].x) and (camp[j+1].y=nuc[i].y)
   then begin
         intersect:=true; xint:=nuc[i].x; yint:=nuc[i].y;
         exit;
        end;
  if (camp[j].x=nuc[i+1].x) and (camp[j].y=nuc[i+1].y)
    then begin
          intersect:=true; xint:=nuc[i+1].x;
          yint:=nuc[i+1].y; exit;
         end;
```

```
{3. Dreapta şi latura nu sunt paralele}
  if (Ad*Bl<>Bd*Al) then
     if Al<>0 then begin
                    yint:=(Ad*Cl-Cd*Al)/(Bd*Al-Ad*Bl);
                    xint:=(-Bl*yint-Cl)/Al;
                   end
     else
          begin yint:=-Cl/Bl;
                xint:=(-Bd*yint-Cd)/Ad;
          end;
  if (((xint>=nuc[i].x) and (xint<=nuc[i+1].x))
     or ((xint>=nuc[i+1].x) and (xint<=nuc[i].x)))
     and(((yint>=nuc[i].y) and (yint<=nuc[i+1].y))
     or ((yint>=nuc[i+1].y) and (yint<=nuc[i].y)))
     then intersect:=true else intersect:=false;
 end;
function sarrus(a,b,c:lat):real;
begin
sarrus:=a.x*b.y+b.x*c.y+a.y*c.x-c.x*b.y-b.x*a.y-c.y*a.x;
end;

procedure cut(j:integer);

{ Procesează latura j a poligonului P}
 var
     al,bl,cl,ad,bd,cd:real;
     inter: array[1..4] of lat;
     index: array[1..4] of integer;
     k,ii,i,nr:integer;
     copy: pol;
begin
 Ad:=camp[j+1].y - camp[j].y; Bd:=camp[j].x-camp[j+1].x;
 Cd:=camp[j+1].x*camp[j].y-camp[j].x*camp[j+1].y;
 nr:=0;
 for i:=1 to nnuc do
  begin
   Al:=nuc[i+1].y -nuc[i].y; Bl:=nuc[i].x-nuc[i+1].x;
   Cl:=nuc[i+1].x*nuc[i].y-nuc[i].x*nuc[i+1].y;
   if intersect(al,bl,cl,ad,bd,cd,i,j) then
```

```
                    begin
                      nr:=nr+1;
                      inter[nr].x:=xint;
                      inter[nr].y:=yint;
                      index[nr]:=i;
                    end;
 end;
if nr>=2 then
    begin
      if sarrus(camp[j],camp[j+1],nuc[index[1]])>=0 then
         begin
          ii:=1;
          copy[1].x:=inter[1].x; copy[1].y:=inter[1].y;
          for k:=index[1]+1 to index[2] do
                 begin
                    inc(ii); copy[ii]:=nuc[k];
                 end;
          inc(ii); copy[ii].x:=inter[2].x;
          copy[ii].y:=inter[2].y;
          nnuc:=ii;
          inc(ii); copy[ii].x:=inter[1].x;
          copy[ii].y:=inter[1].y;
         end
       else
         begin
          ii:=0;
          for k:=1 to index[1] do
               begin inc(ii); copy[ii]:=nuc[k]; end;
          inc(ii);
          copy[ii].x:=inter[1].x;copy[ii].y:=inter[1].y;
          inc(ii);
          copy[ii].x:=inter[2].x;copy[ii].y:=inter[2].y;
             for k:=index[2]+1 to nnuc do
               begin inc(ii); copy[ii]:=nuc[k]; end;
          nnuc:=ii; inc(ii); copy[ii]:=nuc[1]
         end;
         nuc:=copy;
    end
     else
       if sarrus(camp[j],camp[j+1],nuc[1])>=0 then
```

```
                begin
                  writeln(0); close(output); halt;
                end;
 end;

function area(a:pol;n:integer):real;

{aria poligonului simplu}
 var s:real; i:integer;
 begin
     s:=0;
     for i:=1 to n do
        s:=s+(a[i+1].x-a[i].x)*(a[i+1].y+a[i].y)/2;
     area:=s;
 end;

{programul principal}

begin
  assign(input,'arcas.in');
  reset(input);
  assign(output,'arcas.out');
  rewrite(output);
  init;
  for i:=1 to ncamp do
                    cut(i);
  writeln(area(nuc,nnuc));
  close(input);
  close(output);
end.
```

6.9. Cetate



Arheologii din Bitlanda în timpul săpăturilor au descoperit nişte pietre aşezate oarecum straniu. După studiere au ajuns la concluzia ca acestea formează fragmente ale unui zid circular, care înconjura o cetate veche.

Pentru a proteja de turişti şi de curioşi fragmentele descoperite arheologii au decis să înconjoare fragmentele desco-perite cu un gard din plasă metalică. Înconjurarea fiecărui frag-ment este destul de complicată şi puțin comodă, de aceea s-a hotă-rât să se împrejmuiască toate fragmentele împreună.

Cerinţă

Să se scrie un program care să determine lungimea minimă a plasei necesare pentru a îngrădi fragmentele zidului.

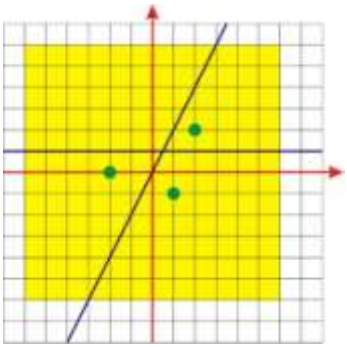
Date de intrare Prima linie a fişierului de intrare conține două numere: **`n (1 ≤ n ≤ 180)`** a fragmentelor de zid şi **`r (1 ≤ r ≤ 100)`** – raza zidului. Urmează **`n`** perechi de numere întregi, care descriu fragmentele de zid: a_i, b_i – mărimea unghiurilor în grade, care descriu începutul şi sfârşitul fragmentului. Unghiurile se măsoară începând cu direcția nord de la centrul cetății, împotriva direcției de mişcare a acelor de ceasornic ($0 \le a_i$; $b_i < 360$, $a_i \ne b_i$). Fiecare fragment este descris la fel împotriva direcției de mişcare a acelor de ceasornic. Fragmentele de zid nu au puncte comune.

Date de ieşire Fişierul de ieşire va conține lungimea minimă a plasei necesare, cu trei cifre după virgulă.

Exemplu

cetate.in	cetate.out
2 10 330 30 90 270	61.888

6.10. Druizi



Ruinele din Stonehenge se află într-un loc special, unde se intersectează M linii energetice ale Pământului. Liniile energetice împart câmpia Stonehenge în sectoare. În fiecare an, în cea mai scurtă zi a anului, N druizi se adună în câmpie pentru un ritual, ce asigură o roadă bogată pentru anul următor. Pentru succesul ritualului este necesar ca în fiecare sector al câmpiei, delimitat de liniile energetice, să se afle nu mai mult decât un druid. Câmpia are forma unui pătrat cu centrul în originea sistemului de coordonate şi are lungimea laturii L. Liniile energetice sunt linii drepte. Druizii sunt punctiformi şi nu se pot afla pe liniile energetice.

Cerinţă

Să se scrie un program care determină dacă există sectoare ale cîmpiei în care se află 2 sau mai mulți druizi.

Date de intrare

Fişierul de intrare conține câteva (cel mult 5) seturi de date, separate prin câte o linie ce conține semnul #

Prima linie a fiecărui set de date conține numerele `N`, `M` şi `L` (`1 ≤ N ≤ 1000, 0 ≤ M ≤ 1000, 1 ≤ L ≤ 2000`).

Urmează `N` linii ce conțin câte 2 numere întregi x_i; y_i — coordonatele druizilor. Toți druizii se află în interiorul câmpiei, ori care doi druizi nu coincid.

Următoarele M linii ale setului de date conțin câte un triplet de numere întregi a_i, b_i, c_i. Numerele corespund coeficienților ecuației $a_ix + b_iy + c_i = 0$, care descrie linia energetică i. Nici un druid nu se află pe o careva linie. Ori care două linii nu coincid. Valorile a_i, b_i, c_i nu depăşesc 10000 după modul.

Date de ieşire

Fişierul de ieşire va conține pentru fiecare set de date cuvîntul YES dacă există cel puțin un sector în care se află doi sau mai mulți druizi, NO, în caz contrar. Fiecare răspuns se va scrie într-o linie aparte, în ordinea apariției seturilor de date în fişierul de intrare.

Exemplu

druizi.in	druizi.out
3 2 3 2 2 1 -1 -2 0 1 1 -1 0 1 -1 # 1 0 100 0 0	YES NO

Soluţie

Pentru implementare vor fi definite tipurile şi structurile:

```
type pt=record x,y,a:real; T:char; end;
     dr=array[1..3003] of pt;
var
   d:dr;
   n,m,i:integer;
   int,xint,r,a,b,c,ae,be,ce,de,
                          l1a,l2a,l1b,l2b,l1c,l2c: real;
```

Pentru citirea şi preprocesarea datelor va fi utilizată procedura readdata:

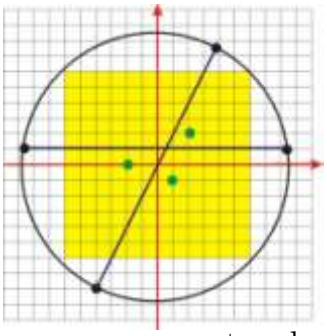
```
procedure readdata;
begin
  readln(n,m,r);
    for i:=1 to n do
       begin readln(d[i].x,d[i].y);  d[i].t:='L'; end;
    for i:=1 to m do
       begin
```

```
            readln(a,b,c);
            if i=1 then begin l1a:=a; l1b:=b; l1c:=c; end;
            if i=2 then begin l2a:=a; l2b:=b; l2c:=c; end;
            ae:=b*b+a*a;
            be:=2*b*c;
            ce:=c*c-a*a*2*r*r;
            de:=be*be-4*ae*ce;
            if a<>0 then begin
                    inc(n);
                    d[n].y:=(-be-sqrt(de))/2/ae;
                    d[n].x:=(-b*d[n].y-c)/a;
                    d[n].t:='D';
                    inc(n);
                    d[n].y:=(-be+sqrt(de))/2/ae;
                    d[n].x:=(-b*d[n].y-c)/a;
                    d[n].t:='D';
                    end
            else begin
                    inc(n);
                    d[n].y:=-c/b;
                    d[n].x:=sqrt(2*r*r-c*c/b/b);
                    d[n].t:='D';
                    inc(n);
                    d[n].y:=d[n-1].y;
                    d[n].x:=-d[n-1].x;
                    d[n].t:='D';
                    end;
        end;
end; {readdata}
```

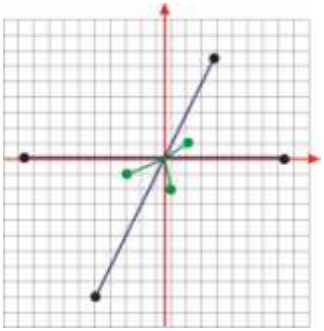
În rezolvare vor fi parcurse următoarele etape:

1. Toate liniile se intersectează într-un punct, care trebuie determinat. Dacă numărul de linii e 0 sau 1, sunt câteva cazuri elementare, care se cercetează aparte. Dacă numărul de linii e mai mare sau egal cu doi, atunci se folosesc oricare 2 linii pentru a determina coordonatele punctului de intersecție.

2. Druizii se află în interiorul pătratului cu latura $2L$ şi centul în originea sistemului de coordonate. Ei se află şi în interiorul cercului cu raza $\sqrt{2}L$ şi centrul în originea sistemului de coordonate. Pentru fiecare din liniile energetice se determină punctele ei de intersecție cu cercul cu raza $\sqrt{2}L$ şi centrul în originea coordonatelor. Aceste puncte se marchează aparținând dreptelor (D) de separare a sectoarelor.



3. Originea sistemului de coordonate se deplasează în punctul de intersecție a dreptelor (liniilor energetice). Apoi se trece la coordonatele polare pentru punctele ce reprezintă druizii şi intersecțiile liniilor energetice cu cercul. (realizare – procedura movecenter)



```
procedure movecenter;
begin
{se determină punctul de intersectie al dreptelor}
if l1a<>0 then
   begin  yint:=(l1c*l2a-l1a*l2c)/(l2b*l1a-l2a*l1b);
          xint:=(-l1b*yint-l1c)/l1a;
   end
else begin  yint:=-l1c/l1b;
            xint:=(-l2b*yint-l2c)/l2a;
     end;
for i:=1 to n do
   begin   {se transferă originea..}
     d[i].x:=d[i].x - xint; d[i].y:=d[i].y - yint;
```

```
{ ... si se determină coordonatele polare - unghiul!}
   if d[i].x<>0 then
        begin
   d[i].a:=180/pi*arctan(abs(d[i].y/d[i].x));
   if (d[i].x<0) and (d[i].y>0) then d[i].a:=180-d[i].a;
   if (d[i].x<0) and (d[i].y<=0) then
                                    d[i].a:=d[i].a+180;
   if (d[i].x>0) and (d[i].y<0) then d[i].a:=360-d[i].a;
        end
           else begin  if d[i].y> 0 then d[i].a:=90;
                       if d[i].y<0 then d[i].a:=270;
                end;
   end
end;  {movecenter}
```

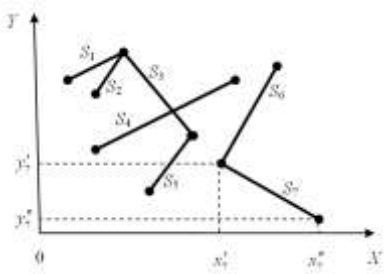
4. Se sortează sistemul de puncte după creşterea unghiului pe care îl formează fiecare punct cu axa *Ox*.

5. Dacă după sortare există cel puţin 2 puncte vecine, reprezentând druizi, rezultă existenţa unui sector al câmpiei cu aceeaşi proprietate.

```
procedure solve;
  var vec: boolean;
begin
{cazurile elementare...}
if ((m=0) and (n=1)) or ( (m=1) and (n=3))
       then writeln('NO');
if ((m=0) and (n>1)) or ((m=1) and (n>4))
       then writeln('YES');
if (m=1) and (n=4)  then
   begin

{se determină pozitia druizilor fata de dreapta}
    if
    sarrus(d[3],d[4],d[1])*sarrus(d[3],d[4],d[2]) < 0
      then writeln('NO') else writeln('YES');
  end;
 if m>=2 then
   begin
{... si cazul general}
```

```
        movecenter;
        qsort(d,1,n);
        vec:=FALSE;
        for i:=1 to n-1 do
        if (d[i].t='L') and (d[i+1].t='L') then
vec:=TRUE;
        if vec then writeln('YES') else writeln('NO');
    end;
end; {solve}
```

6.11 Segmente[15]



Se consideră mulțimea S, formată din n segmente distincte, notate prin S_1, S_2, ..., S_i, ..., S_n. Fiecare segment S_i este definit prin coordonatele întregi carteziene (x_i', x_i'', y_i', y_i'') ale extremităților sale.
Segmentele S_i, S_j se numesc adiacente, dacă ele au o extremitate comună.

Segmentele S_i, S_j se numesc conectate dacă există o succesiune de segmente, care începe cu S_i şi se termină cu S_j şi în care oricare două segmente consecutive sunt adiacente.

De exemplu, segmentele S_1, S_5 de pe figura alăturată sunt conectate întrucât există o succesiune de segmente consecutiv adiacente: S_1, S_3, S_5.

Submulțimea de segmente Q, $Q \subseteq S$, formează o rețea, dacă segmentele respective sunt conectate între ele, fără a avea însă conexiuni cu segmentele din submulțimea $S \setminus Q$.
În general, mulțimea de segmente S poate avea mai multe rețele.

[15] Olimpiada Republicană la Informatică, 2010

De exemplu, mulțimea de segmente de pe figura de mai sus conține 3 rețele: {S_1, S_2, S_3, S_5}, {S_4} şi {S_6, S_7}.

Sarcină. Elaborați un program, care, cunoscând mulțimea de segmente S, calculează numărul de rețele k.

Date de intrare. Fişierul text SEGMENT.IN conține pe prima linie numărul întreg n. Fiecare din următoarele n linii ale fişierului de intrare conține numerele întregi x_i', x_i'', y_i', y_i'' separate prin spațiu. Linia i+1 a fişierului de intrare conține coordonatele extremităților segmentului S_i.

Date de ieşire. Fişierul text SEGMENT.OUT va conține pe singură linie numărul întreg k.

Exemplu.

SEGMENT.IN	SEGMENT.OUT
7 1 7 4 8 2 6 4 8 4 8 7 4 2 4 9 7 5 2 7 4 8 3 11 8 8 3 13 1	3

Restricţii.

- $1 \le n \le 250$; $-32000 \le x_i', x_i'', y_i', y_i'' \le 32000$.
- Timpul de execuție nu va depăşi 1,5 secunde.
- Programul va folosi cel mult 32 Megaocteți de memorie operativă. Fişierul sursă va avea denumirea SEGMENT.PAS, SEGMENT.C sau SEGMENT.CPP.

Rezolvare

Vom construi consecutiv fiecare din eventualele rețele după cum urmează:

1. Inițial, stabilim $Q=\{S_1\}$ şi $S=S/\{S_1\}$.
2. Examinând fiecare segment din mulțimea S, le selectăm pe cele conectate cu segmentele din Q şi le trecem din S în Q. Dacă astfel de segmente nu mai există, s-a obținut o rețea.
3. În continuare, stabilim $Q=\{S_i\}$, unde S_i este unul din segmentele rămase în mulțimea S şi construim din nou o eventuală rețea.
4. Procesul de construire a rețelelor se termină atunci, când mulțimea S devine una vidă.

În programul ce urmează coordonatele extremităților de segmente se memorează în tablourile X1, Y1, X2, Y2. Adiacența segmentelor se verifică cu ajutorul funcției booleene SuntAdiacente, iar conectivitatea segmentelor din mulțime S cu cele din mulțimea Q – cu ajutorul funcției booleene EsteConectat.

```
Program Segmente; { Clasele 07-09 }
const nmax=250; { numarul maximal de segmente }
type Segment=1..nmax;
var Q, S : set of Segment;
    X1, Y1, X2, Y2 : array [1..nmax] of integer;
    n : 1..nmax; { numarul de segmente }
    k : 0..nmax; { numarul de retele }

procedure Citeste;
var i : 1..nmax;
    Intrare : text;
begin
    assign(Intrare, 'SEGMENT.IN');
    reset(Intrare);
    readln(Intrare, n);
    for i:=1 to n do
       readln(Intrare, X1[i], Y1[i], X2[i], Y2[i]);
    close(Intrare);
end; { Citeste }
```

```
procedure Scrie;
var Iesire : text;
begin
    assign(Iesire, 'SEGMENT.OUT');
    rewrite(Iesire);
    writeln(Iesire, k);
    close(Iesire);
end; { Scrie }

function SuntAdiacente(i, j : Segment) : boolean;
{ returneaza true daca segmentele i, j sunt adiacente }
begin
     SuntAdiacente := ((X1[i]=X1[j]) and (Y1[i]=Y1[j]))
     or ((X1[i]=X2[j]) and (Y1[i]=Y2[j])) or
     ((X2[i]=X1[j]) and (Y2[i]=Y1[j])) or ((X2[i]=X2[j])
     and (Y2[i]=Y2[j]));
end; { SuntAdiacente }

function EsteConectat(i : Segment) : boolean;
{ returneaza true daca segmentul i este conectat cu
segmentele din Q }
label 1;
var j : Segment;
begin
    EsteConectat := false;
    for j:=1 to n do
    if j in Q then
             begin
                 if SuntAdiacente(i, j) then
                        begin EsteConectat := true;
                              goto 1;
                        end; { if }
             end; { if }
1: end; { EsteConectat }

procedure NumaraRetele;
label 2;
var i, j : Segment;
    Indicator : boolean;
begin { formam multimea de segmente S }
```

```
    S:=[];
    for i:=1 to n do S:=S+[i];
    k:=0; { numaram retelele }
repeat k:=k+1;
    { includem in multimea Q un segment din S }
    for i:=1 to n do
     if i in S then
                    begin
                      Q:=[i]; S:=S-[i];
                      goto 2;
                    end; { if }
     2: { includem in Q segementele conexe din S }
  repeat
    Indicator:=true;
    for i:=1 to n do
            if i in S then begin
                        if EsteConectat(i) then
                                      begin
                                       Q:=Q+[i];
                                       S:=S-[i];
                                       Indicator:=false;
                                      end; { if }
                           end; { if }
   until Indicator;
until S=[];
end; { NumaraRetele }

begin
    Citeste;
    NumaraRetele;
    Scrie;
end.
```

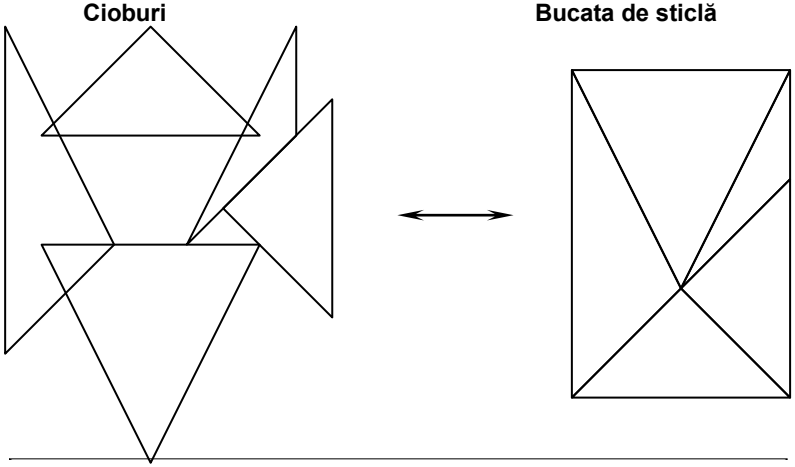
Din analiza procedurii NumaraRetele rezultă că numărul de operații cerut de algoritm este de ordinul n^3. Conform restricțiilor problemei, $1 \leq 250 \leq n$. Prin urmare, numărul necesar de operații va fi de ordinul 10^8, mărime comparabilă cu capacitatea de prelucrare a calculatoarelor din laboratorul de informatică.

6.12 Sticla[16]

Pe o masă se află cîtevă cioburi plate triunghiulare de sticlă (vezi desenul). Se consideră că:

- toate aceste cioburi s-au format dintr-o singură bucată dreptunghiulară de sticlă, marginile căreia, inițial, erau paralele cu marginile mesei;
- toate cioburile s-au format în rezultatul unei singure lovituri, aplicate într-un punct ce nu se afla pe marginile bucății dreptunghiulare de sticlă;
- în momentul formării cîte unul din vîrfurile fiecărui ciob triunghilar se afla în punctul de lovire;
- după formare, unele cioburi puteau să fi fost mutate;
- bucata dreptunghiulară de sticlă poate fi reconstituită prin mutarea paralelă a cioburilor sau, prin alte cuvinte, după lovire nici unul din cioburi nu fost rotit.

Scrieți un program care reconstituie din toate cioburile triunghiulare bucata dreptunghiulară de sticlă.



[16] Olimpiada Republicană la Informatică, 2006

Date de intrare.

Introducem în studiu un sistem de coordonate axele căruia sînt paralele cu marginile mesei. Fiecare ciob va fi definit prin trei perechi de numere ce reprezentă coordonatele vîrfirilor triunghiului respectiv.

Fişierul text STICLA.IN conține pe prima linie numărul întreg N – numărul de cioburi. Fiecare din următoarele N linii ale fişierului de intrare conține cîte trei perechi de numere întregi separate prin spațiu, reprezentînd coordonatele vîrfulrilor fiecărui ciob.

Date de ieşire.

Fişierul text STICLA.OUT va conține N linii. Pe fiecare linie se vor scrie cîte trei perechi de numere separate prin spațiu ce reprezentă coordonatele vîrfurilor ciobului respectiv în momentul cînt el se află în componența bucății dreptunghiulare de sticlă. Bucata dreprunghiulară de sticlă trebuie reconstituită în aşa mod încît coordonatele tuturor vîrfurilor să fie nenegative, iar unul din vîrfuri să se afle în originea sistemului de coordonate. Listarea cioburilor se va face în ordinea apariției lor în fişierul de intrare. Listarea vîrfurilor fiecărui ciob se va face în ordine arbitrară.

Dacă problema admite mai multe soluții, în fişierul de ieşire se va indica doar una din ele.

Exemplu.

STICLA.IN

```
5
0 0 3 3 0 9
1 3 7 3 4 -3
1 6 7 6 4 9
5 3 8 6 8 9
6 4 9 1 9 7
```

STICLA.OUT

```
0 0 3 3 0 9
0 9 6 9 3 3
0 0 6 0 3 3
3 3 6 6 6 9
3 3 6 0 6 6
```

Restricţii. $2 \le N \le 2000$. Coordonatele vîrfurilor fiecărui ciob sînt numere întregi, modulul cărora nu depăşeşte valoarea 10 000. Aria fiecărui ciob este mai mare ca zero. Timpul de

execuţie nu va depăşi 0,2 secunde. Fişierul sursă va avea denumirea STICLA.PAS, STICLA.C sau STICLA.CPP.

Rezolvare

Vom examina două tipuri de triunghiuri: triunghiuri certe şi triunghiuri incerte. Prin definiţie, un triunghi este cert dacă are numai o singură latură paralelă cu una din axele de coordonate. Dacă triunghiul are două laturi paralele cu axele de coordonate, el este incert.

Evident, în cazul triunghiurilor certe vîrful în care a avut loc lovitura se determină în mod univoc ca fiind cel opus laturii paralele cu axa de coordonate.

În cazul unui triunghi incert, există două vîrfuri care ar putea fi ca posibile puncte de lovitură. Întrucît există cel mult 8 triunghiuri incerte, problema poate fi soluţionată prin examinarea tuturor variantelor posibile.

După ce am ales pentru fiecare triunghi care va fi vîrful în care a fost aplicată lovitura, vom deplasa toate triunghiurile în aşa mod încît vîrfurile desemnate ca puncte de lovitură să coincidă. Pentru simplitate, vom amplasa punctul de lovitură în originea sistemului de coordonate.

În continuare verificăm dacă triunghiurile astfel amplasate formează un dreptunghi. În acest scop verificăm respectarea următoarelor condiţii:

1) aria totală a triunghiurilor trebuie să fie egală cu aria dreptunghiului care le înfăşoară;
2) oricare două triunghiuri nu trebuie să aibă intersecţii.

Dacă aceste condiţiile sînt respectate, dreptunghiul din enunţul problemei este construit, fiind necesară doar deplasarea lui în aşa mod ca colţul stînga-jos să fie în originea sistemului de coordonate. Evident, vor fi recalculate şi coordonatele vîrfurilor tuturor dreptunghiurilor.

Pentru a micşora timpul de calcul, vom lua în considerare faptul că oricare două triunghiuri certe nu se intersectează, verificările respective fiind necesare doar pentru triunghiurile incerte.

În programul ce urmează ariile triunghiurilor se calculează cu ajutorul funcției Surf, iar faptul că două triunghiuri se intersectează – cu ajutorul funcției TriunghiuriAuIntersectii.

```
Program Sticla;
type TPoint = record
                x,y:Integer;
              end;

var n:Integer;
type TTriangle = array[1..3] of TPoint;
var P:array[1..2000] of TTriangle;

procedure Citire;
var f:Text;
    i:Integer;
begin
  Assign(f,'sticla.in');
  Reset(f);
  Readln(f,n);
  for i:=1 to n do
    Readln(f,P[i][1].x,P[i][1].y,P[i][2].x,P[i][2].y,
           P[i][3].x,P[i][3].y);
  Close(f);
end;

procedure Afiseaza;
var f:Text;
    i,j:Integer;
    minx, miny: Integer;
begin
    minx:=MAXINT; miny:=MAXINT;
    for i:=1 to n do begin
      for j:=1 to 3 do begin
        if minx>P[i,j].x then minx := P[i,j].x;
```

```
        if miny>P[i,j].y then miny := P[i,j].y;
      end;
    end;

    Assign(f,'sticla.out');
    rewrite(f);
    for i := 1 to n do begin
      for j := 1 to 3 do
         write(f,P[i,j].x-minx,' ',P[i,j].y-miny,' ');
      writeln(f);
    end;
   close(f);
end;

const nIncert:Integer = 0;
{numarul de triunghiuri incerte}
var Incert:array[1..8] of integer;
{indexii triughiurilor incerte}

procedure swap(var a,b:Tpoint);
  var temp:TPoint;
begin
  temp := a;
  a := b;
  b := temp;
end;

procedure TranslateTriangle(index : Integer);
{ misca triughiul astfel incit virful cu indice unu sa
devina (0,0) }
begin
  dec(P[index,2].x,P[index,1].x);
  dec(P[index,3].x,P[index,1].x);
  dec(P[index,2].y,P[index,1].y);
  dec(P[index,3].y,P[index,1].y);
  P[index,1].x := 0; P[index,1].y := 0;
end;

{valorile minimale si maximale pe coordonate pentru
triunghiurile certe}
```

```
const MinCertX : Integer = 0;
const MinCertY : Integer = 0;
const MaxCertX : Integer = 0;
const MaxCertY : Integer = 0;

function Surf(x1,y1,x2,y2,x3,y3:Longint):Longint;
{ intoarce suprafata dubla a unui triunghi }
begin
  Surf := x1*y2 + x2*y3 + x3*y1 -x2*y1 - x3*y2 - x1*y3;
end;

const AriaDubla : LongInt = 0;

procedure DeterminaCorespondenta;
{ Pentru triughiurile certe, determina virful care a
fost in punctul loviturii si il seteaza pe prima
pozitie. Pentru triughiurile incerte, determina
virfurile care sint candidate pentru a fi in punctual
loviturii se le seteaza pe prima si a treia poziti }
var punctTemp : TPoint;
    i,k : Integer;
    Paralel : array[1..3] of Boolean;
begin
     for i:= 1 to n do
     begin
       AriaDubla := AriaDubla +
abs(Surf(P[i,1].x,P[i,1].y,P[i,2].x,P[i,2].y,
        P[i,3].x, P[i,3].y));
       for k:=1 to 3 do
         Paralel[k] := (P[i][k].x=P[i][1+(k mod 3)].x)
             or (P[i][k].y=P[i][1+(k mod 3)].y);
{daca acest triunghi nu e dreptunghic}
if Ord(Paralel[1])+Ord(Paralel[2])+Ord(Paralel[3])=1
         then begin
{du virful opus laturii paralele cu una din axe pr prima
pozitie}
      if Paralel[1] then swap(P[i,1],P[i,3]);
      if Paralel[3] then swap(P[i,1],P[i,2]);
      TranslateTriangle(i);
       for k:=2 to 3 do begin
```

```
        if MinCertX > P[i,k].x then MinCertX := P[i,k].x;
        if MinCertY > P[i,k].y then MinCertY := P[i,k].y;
        if MaxCertX < P[i,k].x then MaxCertX := P[i,k].x;
        if MaxCertY < P[i,k].y then MaxCertY := P[i,k].y;
            end;
          end
        else
           begin
{ daca acest triughi este dreptughic, atunci e posibil
ca lovitura a fost in 2 virfuri }
      inc(nIncert);
      Incert[nIncert] := i;
{ du virful posibil cu indice mai mic pe prima pozitie
si cel mai mare pe ultima pozitie }
            for k:=1 to 3 do
              if Paralel[1+(k mod 3)] then
              begin
                swap(P[i,k],P[i,1]); break;
              end;
            for k:=3 downto 1 do
              if Paralel[1+(k mod 3)] then
              begin
                swap(P[i,k],P[i,3]); break;
              end;
            TranslateTriangle(i);
          end;
     end;
end;

function lower(var p1,p2:TPoint):boolean;
{ verifica daca primul punct e mai jos pe axa unghiulară
decit al 2lea punct }
begin
  lower := -Longint(p2.x)*p1.y+ Longint(p2.y)*p1.x > 0;
end;

function LowerOrSame(var p1,p2:TPoint):boolean;
{ verifica daca primul punct e mai jos sau identic pe
axa unghiulară decit al 2lea punct }
begin
```

```
lowerOrSame:=-Longint(p2.x)*p1.y+Longint(p2.y)*p1.x >=0;
end;

function TriunghiuriAuIntersectii(i,j:Integer):Boolean;
var a,b, u,r : extended;
begin
  { orienteaza pozitiv triughiurile }
  if lower(P[i,3],P[i,2]) then swap(P[i,3],P[i,2]);
  if lower(P[j,3],P[j,2]) then swap(P[j,3],P[j,2]);
  TriunghiuriAuIntersectii:=true;
  if lowerOrSame(P[i,2],P[j,2]) and lower(P[j,2],P[i,3])
then exit;
  if lower(P[i,2],P[j,3]) and lowerOrSame(P[j,3],P[i,3])
then exit;
  if lowerOrSame(P[j,2],P[i,2]) and lower(P[i,2],P[j,3])
then exit;
  if lower(P[j,2],P[i,3]) and lowerOrSame(P[i,3],P[j,3])
then exit;
  TriunghiuriAuIntersectii := false;
end;

function SintCorecte:Boolean;
{ verifica daca triughiurile pot forma un dreptunghi }
var maxx, maxy, minx, miny:Integer;
var i,j,k:Integer;
var x,y:Integer;
begin
  SintCorecte:=False;
  maxx := MaxCertX; maxy := MaxCertY; minx := MinCertX;
miny := MinCertY;
  for i:=1 to nIncert do
    for j:=2 to 3 do
    begin
      if minx>p[Incert[i],j].x then minx :=
p[Incert[i],j].x;
      if miny>p[Incert[i],j].y then miny :=
p[Incert[i],j].y;
      if maxx<p[Incert[i],j].x then maxx :=
p[Incert[i],j].x;
      if maxy<p[Incert[i],j].y then maxy :=
```

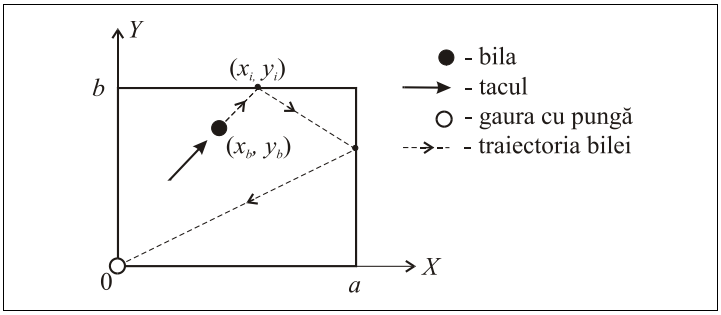
```
p[Incert[i],j].y;
    end;

  { verifica ca bounding box sa fie de dimensiunile
corecte }
  if (2*LongInt(maxx-minx)*LongInt(maxy-
miny)<>AriaDubla) then exit;

  { verifica sa nu fie intersectii intre cele incerte si
oricare alt triunghi }
  for i:=1 to nIncert do
    for k:=1 to n do
      begin
        if k=Incert[i] then continue;
        if TriunghiuriAuIntersectii(k,Incert[i]) then
exit;
      end;
  SintCorecte:=True;
end;

function CautaSolutie(i:Integer):Boolean;
var r:Boolean;
begin
  if (i > nIncert) then begin
    if SintCorecte
    then begin
      Afiseaza;
      CautaSolutie:=True;
      exit;
    end
    else begin CautaSolutie := False; exit; end;
  end;
  if CautaSolutie(i+1) then begin CautaSolutie:=true;
exit; end;

  swap(P[Incert[i],1],P[Incert[i],3]);
  TranslateTriangle(Incert[i]);

  r:=CautaSolutie(i+1);
```

```
  { restabilim datele ca la intrare }
  swap(P[Incert[i],1],P[Incert[i],3]);
  TranslateTriangle(Incert[i]);

  CautaSolutie:=r;
end;

begin
  Citire;
  DeterminaCorespondenta;
  CautaSolutie(1);
end.
```

Din analiza textului programului Sticla se poate deduce că timpul cerut de algoritm este de ordinul $O(n+n\cdot 2^k)$, unde n este numărul de cioburi, iar k este numărul de cioburi cu două laturi paralele la axele de coordonate. Întrucît $k\le 8$, programul se încadrează uşor în timpul alocat.

6.13 Biliard

În cazul *Biliardului Matematic* masa de joc este simbiolizată printr-un dreptunghi cu dimensiunile $a\times b$, cu laturile paralele cu axele de coordonate şi originea sistemului de coordonate în colţul stînga-jos (*Fig. 1*).



Pe masa de joc se află o bilă punctiformă cu coordonatele (x_b, y_b). În colțul stînga-jos masa de biliard are o gaură punctiformă, prin care, în cazul unei lovituri reuşite cu tacul, bila cade într-o pungă. Se consideră că forța de frecare este egală cu zero, iar loviturile bilei de marginile mesei se produc conform legii "unghiul de cădere este egal cu unghiul de reflexie". În *Biliardul Matematic* se consideră reuşită doar acea lovitură cu tacul, după care, înainte de a cădea în pungă, bila se loveşte de marginile mesei exact de k ori, $k \geq 1$. Conform regulilor jocului, colțurile mesei nu aparțin nici la una din margini. Lovitura cu tacul se defineşte prin coordonatele (x_i, y_i) ale punctului de pe marginea mesei, în care bila va lovi pentru prima oară.

Elaborați un program, care, cunoscînd coordonatele (x_b, y_b) şi numărul de lovituri k ale bilei de marginile mesei, calculează toate variantele posibile de lovituri reuşite cu tacul.

Date de intrare. Fişierul text BILIARD.IN conține pe prima linie numerele reale a, b, separate prin spațiu. Linia a doua a fişierului de intrare conține numerele reale x_b, y_b, separate prin spațiu. Linia a treia a fişierului de intrare conține numărul întreg k.

Date de ieşire. Fişierul text BILIARD.OUT va conține pe prima linie un număr întreg n – numărul variantele posibile de lovituri reuşite cu tacul. Fiecare din următoarele n linii ale fişierului de ieşire va conține cîte două numere reale scrise fără specificatori de format şi separate prin spațiu. Linia $i+1$ a fişierului de ieşire va conține coordonatele x_i, y_i ce definesc o lovitură reuşită cu tacul.

Exemplu.

BILIARD.IN

```
3.0 2.0
1.5 1.5
2
```

BILIARD.OUT

```
3
0.0000000000E+00 1.2000000000E+00
1.0909090909E+00 0.0000000000E+00
2.4000000000E+00 2.0000000000E+00
```

Restricţii. $1 \le k \le 20$; $0 \le a, b \le 100$. Timpul de execuție nu va depăşi o secundă. Fişierul sursă va avea denumirea BILIARD.PAS, BILIARD.C sau BILIARD.CPP.

Rezolvare

Pentru a studia mişcarea bilei, introducem în studiu mese virtuale de joc. Prima masă virtuală se obține prin reflexia în "oglindă" a mesei de biliard, oglinda respectivă fiind formată de marginea de care bila s-a lovit pentru prima oară (*Fig. 2*).

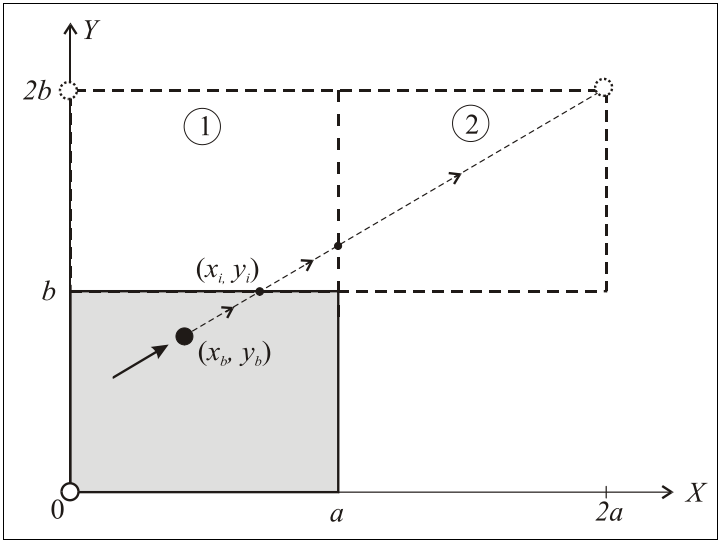


Fig. 2

A doua masă virtuală se obține prin reflexia în "oglindă" a primei mese virtuale, "oglinda" respectivă fiind formată de marginea virtuală de care bila s-a lovit a doua oară. Evident, într-un mod similar, pot fi obținute toate cele k mese virtuale de joc.

Întrucît loviturile bilei de marginile mesei se produc conform legii "unghiul de cădere este egal cu unghiul de reflexie", traectoria bilei pe mesele virtuale reprezintă o linie dreaptă, care reuneşte punctul cu coordonatele (x_b, y_b) cu imaginea în "oglindă" a găurii punctiforme.

Prin urmare, pentru a determina toate variantele posibile de lovituri reuşite cu tacul, divizăm planul cartezian în dreptunghiuri de dimensiunile $a \times b$ şi plasăm în punctele cu coordonatele $(2ia, 2jb)$, $-k \le i, j \le k$, imagini ale găurii punctiforme. În continuare, reunim prin linii drepte punctul (x_b, y_b) cu fiecare din punctele $(2ia, 2jb)$ şi selectăm segmentele care intersectează exact k laturi ale dreptunghiurilor respective. Evident, nu vor fi examinate dreptele care trec prin vîrfuri de dreptunghiuri.

```
Program Biliard;
const kmax=25;
var a, b, { dimensiunile mesei }
    x, y : real; { coordonatele bilei }
    k,    { numarul cerut de lovituri de marginile mesei }
    m : integer;  { numarul de lovituri reusite cu tacul }
    L : array [1..kmax, 1..2] of real; { loviturile
reusite cu tacul}
    Intrare, Iesire : text;

procedure Citeste;
begin
  assign(Intrare,'BILIARD.IN');
  reset(Intrare);
  readln(Intrare, a, b);
  readln(Intrare, x, y);
  readln(Intrare, k);
  close(Intrare);
end; {Citeste }
```

```
procedure Scrie;
var i : integer;
begin
  assign(Iesire,'BILIARD.OUT');
  rewrite(Iesire);
  writeln(Iesire, m);
  for i:=1 to m do
    writeln(Iesire, L[i, 1],' ',L[i, 2]);
  close(Iesire);
end; { Scrie }
procedure LovituriReusite;
var i, j : integer;
    c, d : real; { coordonatele gaurilor virtuale }
    dir : integer;
    t, u : real;
    trece : boolean;
begin
  m:=0; { numarul curent de lovituri reusite cu tacul }
  for i:=-k to k do
    for j:=-k to k do
      begin
        c:=2*i*a; d:=2*j*b;
        if trunc(abs(c-x)/a)+trunc(abs(d-y)/b)=k then
          begin
           { verifica sa nu treaca prin nici un alt virf }
            if c>x then dir:=-1 else dir:=1;
            t:=c+dir*a;
            trece:=false;
            while dir*t<dir*x do
              begin
                u:=y+(d-y)*(t-x)/(c-x);
                u:=u/b;
                if (abs(u-round(u))<0.00001) then
trece:=true;
                t:=t+dir*a;
              end; { while }
            if not(trece) then
              begin
                m:=m+1;
```

```
                if (((b-y)*(c-x)-(a-x)*(d-y))*((b-y)*
                   (0-x)-(a-x)*(b-y))>=0) and
                   (((b-y)*(c-x)-(0-x)*(d-y))*((b-y)*
                   (a-x)-(0-x)*(b-y))>0) then
                  begin L[m, 1]:=x+(c-x)*(b-y)/(d-y);
                        L[m, 2]:=b; end;
                if (((0-y)*(c-x)-(a-x)*(d-y))*((0-y)*
                   (a-x)-(a-x)*(b-y))>=0) and
                   (((b-y)*(c-x)-(a-x)*(d-y))*((b-y)*
                   (a-x)-(a-x)*(0-y))>0) then
                  begin L[m, 1]:=a;
                        L[m, 2]:=y+(d-y)*(a-x)/(c-x); end;
                if (((0-y)*(c-x)-(0-x)*(d-y))*((0-y)*
                   (a-x)-(0-x)*(0-y))>=0) and (((0-y)*
                   (c-x)-(a-x)*(d-y))*((0-y)*(0-x)-
                   (a-x)*(0-y))>0) then
                  begin L[m, 1]:=x+(c-x)*(0-y)/(d-y);
                        L[m, 2]:=0; end;
                if (((0-y)*(c-x)-(0-x)*(d-y))*((0-y)*
                   (0-x)-(0-x)*(b-y))>=0) and (((b-y)*
                   (c-x)-(0-x)*(d-y))*((b-y)*(0-x)-
                   (0-x)*(0-y))>0) then
                  begin L[m, 1]:=0;
                        L[m, 2]:=y+(d-y)*(0-x)/(c-x); end;
                if abs(L[m, 1])+abs(L[m, 2])<0.000001
                then m:=m-1;
              end; { then }
          end; { then }
      end;
end; { LovituriReusite }

begin
  Citeste;
  LovituriReusite;
  Scrie;
end.
```

Numărul de execuții ale instrucțiunii compuse **begin ... end** din componența ciclurilor imbricate **for** ale procedurii LovituriReusite este $4k^2$. Evident, pentru $k \leq 20$, timpul cerut de această procedură va fi cu mult mai mic decît o secundă.

Notaţii

$\overrightarrow{p_1p_2}$	–	segment orientat (vector) cu originea în punctul p_1
$S=\{p_1,...,p_N\}$	–	mulțimea S, cu elementele $p_1,\ldots,p_N$
$\lvert S\rvert$	–	cardinalul mulțimii S
$k\uparrow$	–	incrementarea valorii k cu 1
$k\leftarrow m$	–	variabilei k i se atribuie valoarea variabilei m
$[a,b]$	–	segment cu extremitățile a şi b.
$\triangle$	–	triunghi cu vârfuri în punctele a, b, c
$DV(S)$	–	diagrama Voronoi pentru mulțimea S
$T(S)$	–	triangularizarea mulțimii S
$Q(S)$	–	înfăşurătoarea convexă a mulțimii S
$A\Rightarrow B$	–	din A rezultă B.
$A\Leftrightarrow B$	–	A este echivalent cu B
$x\in X, x\notin X$	–	x aparține X (x nu aparține X)
$\{x\in X:Q\}$	–	submulțimea elementelor x din X, care satisfac condiția Q
$A\subseteq B$	–	A se conține în B (A este submulțime a mulțimii B)

`qsort(A,n)` – apel al procedurii pentru sortarea structurii de date **`A`** din **`n`** elemente.

Bibliografie

1. Ammeraal Leen. *Programming Principles in Computer Graphics*. John Wiley and Sons, 1986.

2. Ammeraal Leen. *Computer Graphics for the IBM PC*. John Wiley and Sons, 1986.

3. Broşura Olimpiadei Republicane la Informatică. Anii 2003 - 2013

4. Cerchez Emanuela. *Programarea în limbajul C/C++ pentru liceu. Vol* III. Polirom, Iaşi, 2007.

5. Corlat Sergiu. *Algoritmi şi probleme de geometrie computațională*. Chişinău, Prut Internațional, 2009

6. Cormen Thomas. *Introducere în algoritmi*. Agora, Cluj.

7. Foley James, Andries Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Publishing, 1984.

8. Giumale Cristian. *Introducere în analiza algoritmilor*. Polirom, Iaşi, 2004.

9. Sedgewick, Th.; *Algorithms in C*. Addison Wesley, 2001

10. Williamson Gill. *Combinatorics for Computer Science*. Dover publications. New York, 2002.

11. Ласло М. *Вычислительная геометрия и компьютерная графика на C++*. Бином, Москва, 1997.

12. Нагао М. *Структуры и базы данных*. Мир, Москва, 1986.

13. Новиков Ф.А. *Дискретная математика для программистов*. Питер, Санкт Петербург, 2001.

14. Пападимитриу Х. *Комбинаторная оптимизация. Алгоритмы и сложность*. Мир, Москва, 1985.

15. Препарата, Ф.; Шеймос, М. *Вычислительная геометрия. Введение*. Москва: Мир, 1989.