Elemente de geometrie computațională

Materialul următor își propune prezentarea modului de rezolvare pentru câteva probleme de geometrie computațională reducând cât mai mult posibil utilizarea funcțiilor de bibliotecă (mai ales a celor ce implică lucru cu numere reale).

Prezentăm, mai întâi, câteva rezultate utile:

1. Determinarea distanței dintre două puncte din plan.

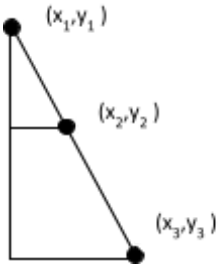
Fie două puncte de coordonate (x_1, y_1), și (x_2, y_2). Distanța în plan dintre ele se calculează folosind teorema lui Pitagora. Se formează un triunghi dreptunghic, în care catetele au lungimi $\left|x_1 - x_2\right|$ respectiv $\left|y_1 - y_2\right|$. Așadar, distanța este lungimea ipotenuzei acestui triunghi, adică:

$$\sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2}$$

Formula este valabilă și pentru cazurile particulare de puncte cu același x sau cu același y.

2. Verificarea coliniarității a trei puncte. Aria unui triunghi determinat de trei puncte în plan.

Fie punctele de coordonate: (x_1, y_1), (x_2, y_2) și (x_3, y_3). Dacă acestea sunt coliniare ne putem imagina următorul desen:



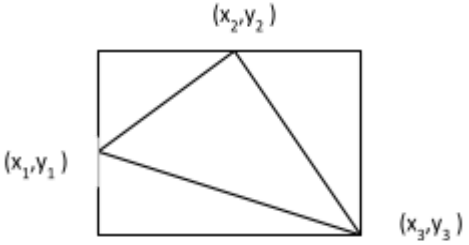
Cele două triunghiuri sunt asemenea, deci avem relațiia: $\frac{\left(x_2 - x_1\right)}{\left(x_3 - x_1\right)} = \frac{\left(y_2 - y_1\right)}{\left(y_3 - y_1\right)}$, care poate fi scrisă:

$\left(x_2 - x_1\right)\left(y_3 - y_1\right) - \left(x_3 - x_1\right)\left(y_2 - y_1\right) = 0$

Aceasta este relația pe care trebuie să o verifice punctele pentru ca ele să fie coliniare.

Un rezultat interesant este următorul: dacă notăm cu D expresia din partea stângă a semnului "=", atunci $\frac{|D|}{2}$ reprezintă aria triunghiului format cu punctele date (am notat cu $|D|$ valoarea absolută a numărului D). Iată o modalitate simplă de justificare. Înscriem triunghiul format de cele 3 puncte într-un dreptunghi, cu laturile

paralele cu axele, ca în figură (consider laturi ale dreptunghiului cele cu x minim, x maxim, y minim, y maxim).



Putem calcula aria triunghiului în următorul mod: Scădem din aria dreptunghiului ariile a trei triunghiuri dreptunghice. Pe exemplul prezentat obținem:

$$\left(x_3 - x_1\right)\left(y_2 - y_3\right) - \frac{\left(x_2 - x_1\right)\left(y_2 - y_1\right)}{2} - \frac{\left(x_3 - x_2\right)\left(y_2 - y_3\right)}{2} - \frac{\left(x_3 - x_1\right)\left(y_1 - y_3\right)}{2}.$$

Desfăcând parantezele și reducând termenii asemenea, vom obține același lucru (eventual cu semne schimbate). Exemplul este unul particular, dar indiferent de poziționarea varfurilor triunghiului in raport cu dreptunghiul, relația este valabila (de exemplu, putem avea două vârfuri ale triunghiului suprapuse cu două ale dreptunghiului, vecine sau diagonal opuse).

3. Ecuația dreptei în plan.

Pentru 3 valori date a, b și c, toate punctele din plan care verifică relația $a{\cdot}x + b{\cdot}y + c = 0$, se găsesc pe aceeași dreaptă. Dându-se două puncte (x_1, y_1) și (x_2, y_2), pentru a scrie ecuația dreptei determinată de ele facem următorul raționament. Fie (x, y) un alt punct aflat pe dreapta determinată de cele două puncte date. Din cele de mai sus, înseamnă că el verifică relația: $\left(x_2 - x_1\right)\left(y - y_1\right) - \left(x - x_1\right)\left(y_2 - y_1\right) = 0$. Efectuând operațiile și cuplând altfel termenii obținem: $\left(y_1 - y_2\right)x + \left(x_2 - x_1\right)y + x_1\left(y_2 - y_1\right) - y_1\left(x_2 - x_1\right) = 0$, deci:

$a = y_1 - y_2$

$b = x_2 - x_1$

$c = x_1\left(y_2 - y_1\right) - y_1\left(x_2 - x_1\right)$

Odată obținute a, b, c ca mai sus, așa cum am spus, pentru toate punctele (x, y) de pe dreaptă, expresia $a{\cdot}x + b{\cdot}y + c$ are valoarea 0. Mai mult, este de o mare valoare următorul rezultat: toate punctele din același semiplan, introduse în ecuația dreptei, fac ca expresia să dea o valoare nenulă și cu același semn. Adică pentru toate punctele (x, y) dintr-un semiplan avem: $a{\cdot}x + b{\cdot}y + c > 0$ și pentru toate punctele (x, y) din celălalt semiplan avem $a{\cdot}x + b{\cdot}y + c < 0$.

Aplicații rezolvate

1. Verificare dacă un punct se găsește pe un segment.

Enunț

Cerința

Se dau un punct și un segment în plan. Să se verifice dacă punctul se găsește pe segment.

Date de intrare
Fișierul `punctsegment.in` conține pe prima linie `6` numere naturale separate prin spații, respectiv: X_1, Y_1, X_2, Y_2, X_3, Y_3. Segmentul are capetele (X_2, Y_2) și (X_3, Y_3).

Date de ieșire
Fișierul `punctsegment.out` conține pe primul rând `DA` (dacă punctul de coordonate X_1, Y_1 se găsește pe segment) sau `NU` (în caz contrar).

Restricții și precizări
Numerele din fișierul de intrare sunt întregi cuprinse între `-1001` și `1001`
Segmentul are lungimea nenulă

Exemplu

```
punctsegment.in
2 2 1 1 3 3
punctsegment.out
DA
```

Descrierea soluției

Verificăm mai întâi dacă punctul (X_1, Y_1) se găsește pe dreapta determinată de punctele (X_2, Y_2) și (X_3, Y_3). Acest lucru este necesar dar nu suficient. Trebuie în plus ca X_1 să aparțină intervalului [`minim(`X_2`, `X_3`), maxim(`X_2`, `X_3`)`] și Y_1 să aparțină intervalului [`minim(`Y_2`, `Y_3`),maxim(`Y_2`, `Y_3`)`]. Dacă sunt puse ambele condiții se rezolvă și cazurile particulare când dreapta suport a segmentului este paralelă cu una dintre axele sistemului de coordonate.

Sursa

```
#include <fstream>
#include <stdlib.h>
using namespace std;
ifstream fin ("punctsegment.in");
ofstream fout("punctsegment.out");
int X1, Y1, X2, Y2, X3, Y3;
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
    int d = det(x1, y1, x2, y2, x3, y3);
    if (d!=0)
        return 0;
    if (x1 == x3 && y1 == y3)
        return 1;
    if (x2 == x3 && y2 == y3)
        return 1;
    if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
        return 1;
    else
        return 0;
}
void msgAndOut(char *msg) {
    fout<<msg;
```

```
    exit(0);
}
int main() {
    fin>>X1>>Y1>>X2>>Y2>>X3>>Y3;
    if (punctPeSegment(X3, Y3, X2, Y2, X1, Y1))
        msgAndOut("DA\n");
    else
        msgAndOut("NU\n");
    return 0;
}
```

2. Verificarea intersecției a două segmente.

Enunț

Cerința

Se dau două segmente în plan, specificate prin coordonatele capetelor. Să se verifice dacă au cel puțin un punct comun.

Date de intrare
Fișierul `intersectiesegmente.in` conține pe prima linie `8` numere naturale separate prin spații, respectiv: X_1, Y_1, X_2, Y_2, X_3, Y_3, X_4, Y_4. Primul segment are capetele (X_1, Y_1) și (X_2, Y_2).

Date de ieșire
Fișierul `intersectiesegmente.out` conține pe primul rând `DA` (când segmentele se intersectează) sau `NU` (în caz contrar).

Restricții și precizări
Numerele din fișierul de intrare sunt întregi cuprinse între `-1001` și `1001`.
Ambele segmente au lungimea nenulă.

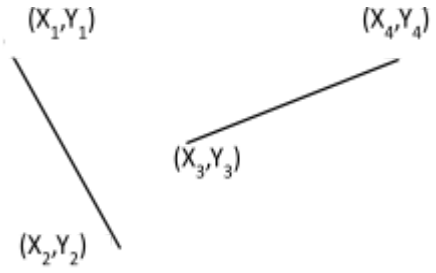
Exemplu

```
intersectiesegmente.in
-1 -1 1 1 -1 1 1 -1
intersectiesegmente.out
DA
```

Descrierea soluției

Pentru a elimina diverse cazuri particulare, putem mai întâi testa următoarele:
- dacă un punct al unui segment coincide cu un punct al celuilalt segment;
- dacă un punct al unui segment se află pe celălalt segment;

Rămâne de tratat cazul general. Introducând coordonatele punctelor unui segment în ecuația dreptei determinate de celălalt segment trebuie să obținem semne contrare. Nu este suficient să facem testul doar pentru unul dintre segmente, ci pentru ambele (vezi figura).



Se observă că deși punctele (X_1, Y_1) și (X_2, Y_2) introduse în ecuația dreptei determinată de celelalte două puncte dau semne contrare, segmentele nu se intersectează, fiind necesar ca și punctele (X_3, Y_3) respectiv (X_4, Y_4) introduse în ecuația dreptei determinată de (X_1, Y_1) și (X_2, Y_2) să dea semne contrare.

Sursa

```
#include <fstream>
#include <stdlib.h>
using namespace std;
ifstream fin ("intersectiesegmente.in");
ofstream fout("intersectiesegmente.out");
int X1, Y1, X2, Y2, X3, Y3, X4, Y4;
int d1, d2, d3, d4;
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
    int d = det(x1, y1, x2, y2, x3, y3);
    if (d!=0)
        return 0;
    if (x1 == x3 && y1 == y3)
        return 1;
    if (x2 == x3 && y2 == y3)
        return 1;
    if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
        return 1;
    else
        return 0;
}
void msgAndOut(char *msg) {
    fout<<msg;
    exit(0);
}
int main() {
    fin>>X1>>Y1>>X2>>Y2>>X3>>Y3>>X4>>Y4;
    if (punctPeSegment(X1, Y1, X2, Y2, X3, Y3)) {
        msgAndOut("DA\n");
    }
    if (punctPeSegment(X1, Y1, X2, Y2, X4, Y4)) {
        msgAndOut("DA\n");
    }
    if (punctPeSegment(X3, Y3, X4, Y4, X1, Y1)) {
        msgAndOut("DA\n");
    }
    if (punctPeSegment(X3, Y3, X4, Y4, X2, Y2)) {
        msgAndOut("DA\n");
    }
    d1 = det(X3, Y3, X4, Y4, X1, Y1);
    d2 = det(X3, Y3, X4, Y4, X2, Y2);
    d3 = det(X1, Y1, X2, Y2, X3, Y3);
    d4 = det(X1, Y1, X2, Y2, X4, Y4);
    if (d1 * d2 < 0 && d3 * d4 < 0)
        msgAndOut("DA\n");
    else
        msgAndOut("NU\n");
    return 0;
}
```

În continuare, s-ar putea formula și cerința: care sunt coordonatele punctului de intersecție al dreptelor suport pentru cele două segmente ?

Pentru rezolvarea acestei cerințe scriem mai întâi ecuațiile celor două drepte și vom obține:

$a_1x + b_1y + c_1 = 0$(pentru dreapta determinată de punctele X_1, Y_1 și X_2, Y_2) respectiv

$a_2x + b_2y + c_2 = 0$ (pentru dreapta determinată de punctele X_3, Y_3 și X_4, Y_4).

Dacă o ecuație este combinație liniară a celeilalte (adică există o valoare nenulă k așa încât $a_1 = k * a_2$, $b_1 = k * b_2$ și $c_1 = k * c_2$), atunci cele două drepte coincid și putem spune că au o infinitate de puncte în comun. Exemplu: $2x + 3y + 4 = 0$ respectiv $6x + 9y + 12 = 0$.
Dacă există k nenul încât $a_1 = k * a_2$, $b_1 = k * b_2$ și $c_1 \neq k * c_2$ atunci cele două drepte nu coincid dar sunt paralele. Exemplu: $2x + 3y + 4 = 0$ respectiv $6x + 9y + 11 = 0$.
În celelalte cazuri cele două drepte au un singur punct de intersecție pe care îl determinăm ca fiind soluția unică a sistemului de ecuații:

$$a_1x + b_1y + c_1 = 0$$

$$a_2x + b_2y + c_2 = 0$$

Un mod de rezolvare este prezentat în continuare. Presupunem că înmulțim prima ecuație cu a_2 și pe a doua cu $-a_1$.

$$+ a_2a_1x + a_2b_1y + a_2c_1 = 0$$

$$- a_1a_2x - a_1b_2y - a_1c_2 = 0$$

Adunând cele două relații observăm că se reduc termenii ce îl conțin pe x și îl putem exprima pe y:

$$y = (a_1c_2 - a_2c_1) / (a_2b_1 - a_1b_2)$$

De fapt doar această formulă ne este necesară în calcule. Observăm că numitorul nu poate fi `0` (din relațiile de calcul pentru coeficientii a și b rezultă că numitorul ar fi `0` doar dacă cele două drepte sunt paralele sau coincid). Pe x îl putem calcula cu relația $x = (-b_1y - c_1)/a_1$. Cazul în care a_1 este `0` corespunde cazului particular că prima dreaptă este paralelă cu axa ox. În acest caz îl extragem pe x din a doua ecuație (nu pot fi simultam `0` a_1 și a_2 căci ar însemna că dreptele sunt paralele sau coincid).

3. Determinarea distanței de la un punct la o dreaptă.

Enunț

Cerința

Se dau în plan, un punct și o dreaptă. Să se determine distanța de la punct la dreaptă.

Date de intrare
Fișierul `distantapunctdreapta.in` conține pe prima linie `6` numere naturale separate prin spații, respectiv: X_1, Y_1, X_2, Y_2, X_3, Y_3. Se cere determinarea distanței de la punctul de coordonate (X_1, Y_1) la dreapta care trece prin punctele de coordonate (X_2, Y_2) și (X_3, Y_3).

Date de ieșire
Fișierul `distantapunctdreapta.out` conține pe primul rând un număr real cu exact două zecimale exacte (fără rotunjire).

Restricții și precizări

Numerele din fișierul de intrare sunt întregi cuprinse între `-1001` și `1001`.
Punctele care determină dreapta sunt distincte.

Exemplu

```
distantapunctdreapta.in
0 1 0 0 1 0
distantapunctdreapta.out
1.00
```

Descrierea soluției

Vom exprima în două moduri aria triunghiului determinat de cele trei puncte.

- prin formula dedusă la începutul prezentării: $\left|\left(X_2 - X_1\right)\left(Y_3 - Y_1\right) - \left(X_3 - X_1\right)\left(Y_2 - Y_1\right)\right|/2$;
- prin arhicunoscuta formulă `baza * înălțimea / 2` ;

În cazul nostru, baza este chiar distanța dintre punctele (X_2, Y_2) și (X_3, Y_3) iar înălțimea este chiar valoarea care ni se cere. Așadar obținem rezultatul:

$$\frac{\left(X_2-X_1\right)\left(Y_3-Y_1\right)-\left(X_3-X_1\right)\left(Y_2-Y_1\right)}{\sqrt{\left(X_3-X_2\right)^2+\left(Y_3-Y_2\right)^2}}$$

Se păstrează valoarea absolută a rezultatului.
Dacă avem deja scrisă ecuația dreptei la care avem de calculat distanța ca fiind: $ax + by + c = 0$, atunci formula se mai poate scrie:

$$\frac{\left|aX_1+bY_1+c\right|}{\sqrt{a^2+b^2}}$$

Sursa

```
#include <fstream>
#include <cmath>
#include <iomanip>
using namespace std;
ifstream fin ("distantapunctdreapta.in");
ofstream fout("distantapunctdreapta.out");
int X1, Y1, X2, Y2, X3, Y3;
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int distantaPuncte(int X1, int Y1, int X2, int Y2) {
    return (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2);
}
int main() {
    fin>>X1>>Y1>>X2>>Y2>>X3>>Y3;
    int a = det(X1, Y1, X2, Y2, X3, Y3);
    if (a < 0)
        a = -a;
    int d = distantaPuncte(X2, Y2, X3, Y3);
    double h = a/sqrt(d);
    fout<<setprecision(2)<<fixed<< (int)(h*100)/100.0;
    return 0;
}
```

4. Determinarea distanței de la un punct la un segment.

Enunț

Cerința

Se dau în plan, un punct și un segment. Să se determine distanța minimă de la punctul dat la un punct aparținând segmentului.

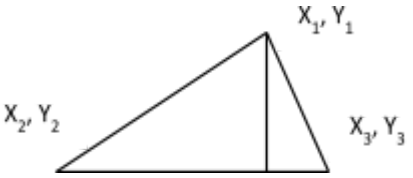
Date de intrare
Fișierul distantapunctsegment.in conține pe prima linie 6 numere naturale separate prin spații, respectiv: X_1, Y_1, X_2, Y_2, X_3, Y_3. Se cere determinarea distanței minime de la punctul de coordonate (X_1, Y_1) la un punct aparținând segmentului cu capetele în punctele (X_2, Y_2) și (X_3, Y_3).

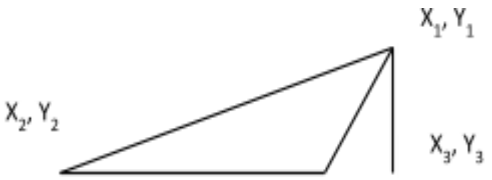
Date de ieșire
Fișierul distantapunctsegment.out conține pe primul rând un număr real cu exact două zecimale (fără rotunjire).

Restricții și precizări
Numerele din fișierul de intrare sunt întregi cuprinse între −1001 și 1001.
Punctele care determină segmentul sunt distincte.

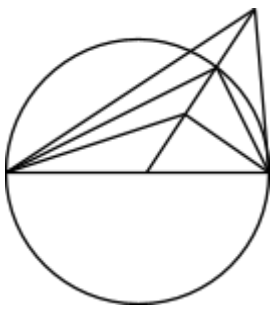
Exemplu
```
distantapunctsegment.in
0 1 0 0 1 0
distantapunctsegment.out
1.00
```

Descrierea soluției
Dacă piciorul perpendicularei dusă din punct pe dreapta suport a segmentului cade chiar pe segment, problema se reduce la cea anterioară. Altfel, distanța de la punct la segment este egală cu distanța de la punct la unul dintre capetele segmentului.
Testăm mai întâi dacă punctul este chiar pe segment, caz în care rezultatul este 0.
Apoi pentru a decide dacă piciorul perpendicularei cade chiar pe segment facem observația: considerând triunghiul format de cele trei puncte, piciorul perpendicularei dusă din (X_1, Y_1) pe dreapta determinată de punctele (X_2, Y_2) și (X_3, Y_3) este pe segment dacă atât unghiul din (X_2, Y_2) cât și cel din (X_3, Y_3) sunt mai mici sau egale cu 90 de grade (vezi figurile următoare).





Pentru a verifica dacă un unghi este ascuțit sau obtuz, ne putem folosi de teorema lui Pitagora, așa cum se observă din figura următoare:



Considerăm cele trei triunghiuri cu o latură ca fiind diametrul desenat și cu cel de-al treilea vârf așa cum se vede în figură. Triunghiul care are al treilea vârf pe cerc are acel unghi de 90 de grade. Suma pătratelor laturilor sale (altele decât cea comună cu diametrul) este egală cu pătratul diametrului. Triunghiul care are al treilea vârf în interiorul cercului (și acel unghi este mai mare decât 90 de grade) are suma pătratelor laturilor (altele decât cea comună cu diametrul) mai mică decât pătratul diametrului iar pentru triunghiul cu vârful în afara cercului (deci cu acel unghi ascuțit), suma pătratelor laturilor (altele decât cea comună cu diametrul) este mai mare decât pătratul diametrului.
Așadar, pentru a testa dacă unghiul este obtuz putem verifica inegalitatea (din al treilea caz) din teorema lui Pitagora.

Sursa

```
#include <fstream>
#include <cmath>
#include <iomanip>
using namespace std;
ifstream fin ("distantapunctsegment.in");
ofstream fout("distantapunctsegment.out");
int X1, Y1, X2, Y2, X3, Y3;
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int distantaPuncte(int X1, int Y1, int X2, int Y2) {
    return (X1-X2)*(X1-X2) + (Y1-Y2)*(Y1-Y2);
}
int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
    int d = det(x1, y1, x2, y2, x3, y3);
    if (d!=0)
        return 0;
    if (x1 == x3 && y1 == y3)
        return 1;
    if (x2 == x3 && y2 == y3)
        return 1;
    if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
        return 1;
    else
        return 0;
}
int main() {
    fin>>X1>>Y1>>X2>>Y2>>X3>>Y3;
    int a = det(X1, Y1, X2, Y2, X3, Y3);
    if (a == 0) {
        if (punctPeSegment(X3, Y3, X2, Y2, X1, Y1)) {
            fout<<setprecision(2)<<fixed<< 0.0;
        } else {
            int d1 = distantaPuncte(X1, Y1, X2, Y2);
            int d2 = distantaPuncte(X1, Y1, X3, Y3);
            if (d1 < d2)
                fout<<setprecision(2)<<fixed<< (int)(sqrt(d1)*100)/100.0;
            else
                fout<<setprecision(2)<<fixed<< (int)(sqrt(d2)*100)/100.0;
        }
        return 0;
    }
    if (a < 0)
        a = -a;
    int d1 = distantaPuncte(X2, Y2, X3, Y3);
    int d2 = distantaPuncte(X1, Y1, X3, Y3);
    int d3 = distantaPuncte(X2, Y2, X1, Y1);
    if (d2 < d1 + d3 && d3 < d1 + d2) {
        double h = a/sqrt(d1);
        fout<<setprecision(2)<<fixed<< (int)(h*100)/100.0;
    } else {
        int d1 = distantaPuncte(X1, Y1, X2, Y2);
        int d2 = distantaPuncte(X1, Y1, X3, Y3);
        if (d1 < d2)
            fout<<setprecision(2)<<fixed<<(int)(sqrt(d1)*100)/100.0;
        else
            fout<<setprecision(2)<<fixed<<(int)(sqrt(d2)*100)/100.0;
    }
    return 0;
}
```

5. Determinarea ariei unui poligon simplu (ale cărui laturi nu se autointersectează).

Enunț

Cerința

Se dau coordonatele în plan pentru `n` puncte. Să se afișeze valoarea ariei poligonului pe care acestea îl formează.

Date de intrare

Fișierul ariapoligonsimplu.in conține pe prima linie numărul de vârfuri ale poligonului, notat cu `n`. Pe următoarele `n` linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui vârf. Acestea sunt date într-un sens de parcurgere a laturilor poligonului.

Date de ieșire

Fișierul `ariapoligonsimplu.out` conține pe primul rând un numpăr natural, cu exact o zecimală, reprezentând valoarea cerută.

Restricții și precizări

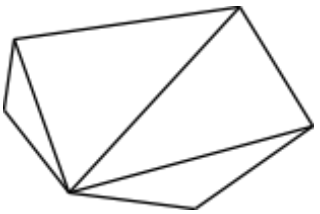
Numerele din fișierul de intrare sunt întregi cuprinse între `-1001` și `1001`
Poligonul nu este neapărat convex dar nu se autointersectează.

Exemplu

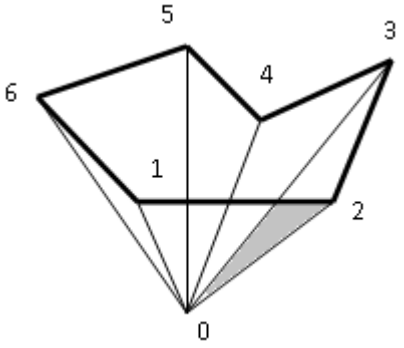
```
ariapoligonsimplu.in
4
0 0
1 0
1 1
0 1
ariapoligonsimplu.out
1.0
```

Descrierea soluției

Să analizăm mai întâi cazul poligonului convex. Putem alege un punct în interiorul său (care poate fi și unul dintre vârfuri) și însumăm ariile triunghiurilor formate cu acel punct și oricare două vârfuri vecine ale poligonului (vezi figura).



Pentru cazul general pare că acest lucru nu mai este valabil. Însă vom vedea că lucrurile sunt aproape la fel de simple. Să analizăm mai întâi figura următoare:



Vom calcula valoarea D (cu semn) pentru prmătoarele triplete de puncte (0,1,2), (0,2,3), (0,3,4), (0,4,5), (0,5,6), (0,6,1). Adică vom considera triunghiurile formate de punctul 0 cu fiecare latură a poligonului, dar vom păstra un sens de parcurgere (coordonatele punctelor unui triplet se vor scrie în formulă în ordinea dată). Pentru triunghiul haşurat observăm că aria sa se adună cu un semn în cazul scrierii lui D pentru triunghiul (0, 1, 2) şi cu celălalt semn în cazul scrierii lui D pentru triunghiul (0, 2, 3). Aceasta pentru că punctul 0 este de o parte a dreptei (1, 2) în sensul de orientare a sa de la 1 la 2, dar este de cealaltă parte a dreptei (2, 3) dacă o privim orientartă de la 2 la 3. Raţionamentul este valabil pentru toare zonele exterioare poligonului, deci valoarea ariei lor se va anuala. Pentru zonele interioare valoarea se va aduna o singură dată şi întotdeauna cu acelaşi semn. Aşadar, rezultatul este:

$$\frac{\left|D_{(0,1,2)}+D_{(0,2,3)}+\ldots+D_{(0,n-1,n)}+D_{(0,n,1)}\right|}{2}$$

Sursa

```
#include <fstream>
#include <iomanip>
#define DIM 100010
using namespace std;
ifstream fin ("ariapoligonsimplu.in");
ofstream fout("ariapoligonsimplu.out");
pair<int, int> v[DIM];
int sol;
int n, i;
int aria(pair<int, int> a, pair<int, int> b, pair<int, int> c) {
    return  (b.first-a.first) * (c.second-a.second) -
            (c.first-a.first) * (b.second-a.second);
}
int main() {
    fin>>n;
    for (i=1;i<=n;i++) {
        fin>>v[i].first>>v[i].second;
    }
    v[0] = v[n];
    for (i=0;i<n;i++) {
        sol += aria(v[0], v[i], v[i+1]);
    }
    fout<<setprecision(1)<<fixed<<sol/2.0;
    return 0;
}
```

6. Ordonarea unor puncte date în plan după unghiul pe care segmentul ce le uneşte cu originea îl face cu axa ox.

Enunţ

Cerinţa

Se dau puncte distincte în plan. Asociem fiecărui punct semidreapta care porneşte din originea sistemului de coordonate şi trece prin acel punct. Să se afişeze punctele în ordine crescătoare a unghiului pe care semidreapta asociată îl face cu semidreapta dusă către plus infinit, a axei ox. Dacă două unghiuri sunt egale se va afişa punctul cel mai apropiat de origine.

Date de intrare

Fişierul `sortareunghi.in` conţine pe prima linie un număr `n`, reprezentând numărul de puncte. Pe următoarele `n` linii se găsesc câte două numere separate printr-un spaţiu, reprezentând abscisa respectiv ordonata câte unui punct.

Date de ieşire

Fișierul sortareunghi.out conține `n` linii cu câte două numere separate prin spațiu fiecare, reprezentând abscisa respectiv ordonata câte unui punct, în ordinea cerută.

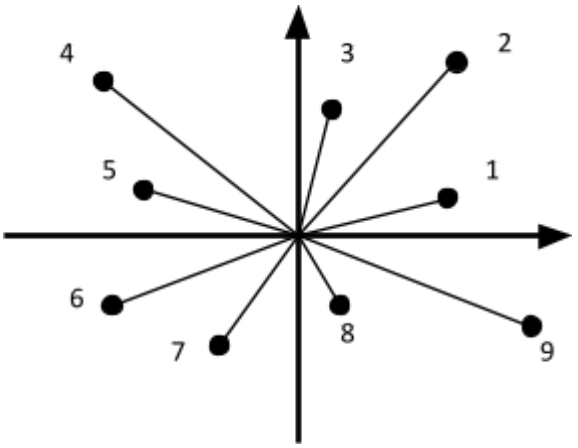
Restricții și precizări
$1 \leq n \leq 100$
Numerele din fișierul de intrare sunt întregi cuprinse între `-1001` și `1001`.
Unghiurile sunt în intervalul `[0,360)`
Punctul `0,0` nu se găsește în fișierele de intrare

Exemplu

```
sortareunghi.in
3
1 1
-1 -1
-1 1
sortareunghi.out
1 1
-1 1
-1 -1
```

Descrierea soluției

Să analizăm figura în care am numerotat punctele chiar cu poziția pe care o ocupă în șirul final.



Observăm că dacă am scrie $D_{(0,1,2)}$ și $D_{(0,2,2)}$ am obține semne diferite. Dacă șirul ar fi sortat vom avea deci mereu același semn pentru $D_{(0,i,i+1)}$. Semnul lui D este deci criteriul pe care îl considerăm la compararea a `2` puncte în funcția de sortare. Apar însă probleme dacă punctele sunt coliniare. În acest caz valoarea D este `0` și când punctele sunt în același cadran, și că sunt în cadrane diferite, deci criteriul semnului lui D nu se mai poate aplica. Această problemă se rezolvă considerând ca prim criteriu de sortare cadranul în care se află cele două puncte de comparat. Astfel, dacă $X > 0$ și $Y \geq 0$ considerăm cadranul `1`, pentru $X \leq 0$ și $Y > 0$. considerăm cadranul `2`, pentru $X < 0$ și $Y \leq 0$ considerăm cadranul `3` iar pentru $X \geq 0$ și $Y < 0$ cadranul `4`. Evident, al treilea criteriu de sortare, ca prioritate este distanța față de origine, conform cerinței.
O altă observație utilă pentru reducerea calculelor este că dacă unul dintre cele `3` puncte pentru care calculăm valoarea D este originea, formula D devine: $x_2y_1 - x_1y_2$. În practică sunt multe cazuri în care toate punctele de sortat sunt în același cadran sau toate deasupra axei ox. Este suficient ca și criteriu de comparare semnul expresiei scrise anterior.

Sursa

```
#include <fstream>
#include <algorithm>
using namespace std;
ifstream fin ("sortareunghi.in");
ofstream fout("sortareunghi.out");
pair<int, int> v[103];
int n, i;
int cadran(int x, int y) {
    if (x > 0 && y >= 0)
        return 1;
    if (x >= 0 && y<0)
        return 4;
    if (y > 0 && x <= 0)
        return 2;
    return 3;
}
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int cmp(const pair<int, int> &a, const pair<int, int> &b) {
    int c1 = cadran(a.first, a.second);
    int c2 = cadran(b.first, b.second);
    if (c1 != c2)
        return c1 < c2;
    else {
        int d = det(0, 0, a.first, a.second, b.first, b.second);
        if (d != 0)
            return d > 0;
        else
            return a.first*a.first+a.second*a.second<b.first*b.first+b.second*b.second;
    }
}
int main() {
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i].first>>v[i].second;
    sort(v+1, v+n+1, cmp);
    for (i=1;i<=n;i++)
        fout<<v[i].first<<" "<<v[i].second<<"\n";
    return 0;
}
```

7. Înfăşurătoarea convexă (cu număr maxim posibil de puncte pe margine).

Enunț

Cerința

Se dau puncte distincte în plan. Să se determine un poligon de arie maximă care are vârfuri dintre punctele date.

Date de intrare

Fișierul infasuratoareconvexa.in conține pe prima linie un număr n, reprezentând numărul de puncte. Pe următoarele n linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui punct.

Date de ieșire

Fișierul infasuratoareconvexa.out conține pe prima linie un număr k, reprezentând numărul de vârfuri ale poligonului determinat. Pe următoarele k linii se găsesc câte două numere separate printr-un spațiu, reprezentând respectiv abscisa și ordonata câte unui punct.

Restricții și precizări

$1 \leq n \leq 100$
Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001
Primul punct care se va afișa va fi cel cu ordonata minimă, iar în caz de egalitate cel cu abscisa minimă
Punctele se vor afișa în sens trigonometric al parcurgerii lor pe înfășurătoare
Se cere soluția cu număr maxim de puncte (pot fi puncte coliniare pe înfășurătoare)

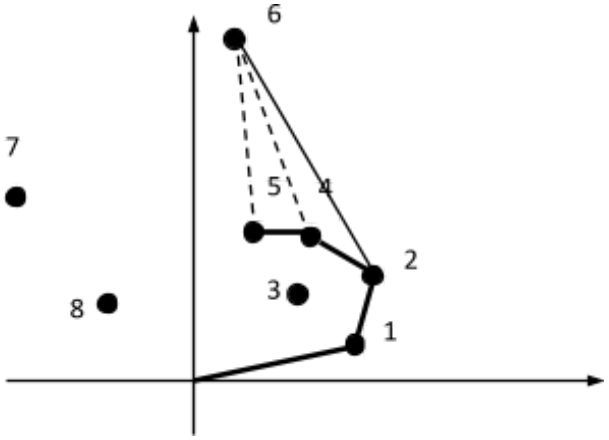
Exemplu

```
infasuratoareconvexa.in
5
1 1
0 0
0 2
2 2
2 0

infasuratoareconvexa.out
4
0 0
2 0
2 2
0 2
```

Descrierea soluției

În prima etapă determinăm punctul cu valoarea Y minimă, iar dintre acestea pe acela cu valoarea X minimă. Scăzând coordonatele sale din coordonatele celorlalte puncte translatăm practic originea sistemului de axe în acel punct și toate celelalte se vor afla în cadranele 1 și 2. Ordonăm apoi celelalte punctele ca în problema anterioară. Originea precum și primul și ultimul punct din șirul ordonat vor face parte sigur din înfășurătoare. Pentru început inițializăm șirul punctelor de pe înfășurătoare cu primele două (originea și primul punct din șirul sortat). Parcurgem restul de puncte în ordinea sortării.



Cu punctele parcurse deja avem construită înfășurătoarea. În figură este reprezentată starea de înaintea analizării punctului 6. Cu punctele de la 1 la 5 este marcată îngroșat înfășurătoarea. Deci șirul punctelor de pe înfășurătoare este 0, 1, 2, 4, 5 (punctul 3 este eliminat). La analizarea punctului 6 observăm că nu putem extinde înfășurătoarea curentă întrucât unghiul care se formează între ultimele 2 puncte din șir (4 și 5) și punctul 6 este unul obtuz (identificăm asta prin semnul lui D pentru cele 3 puncte). Așadar, punctul 6 face să dispară din șir punctul 5. Pe același principiu, punctul 6 face să dispară și punctul 4 (unghiul format cu ultimele două puncte, acum 2 și 4 și punctul 6 este tot obtuz). Însă, unghiul format din 1,2 și 6 este

ascuțit, așadar 6 nu mai elimină puncte din șir iar la final se adaugă el în șir (ultimul punct ca unghi este mereu pe înfășurătoare). O altă observație: cum punctele 0 și 1 fac sigur parte de pe înfășurătoare, mereu vor fi în șir cel puțin 2 puncte.

Șirul punctelor aflate pe parcurs pe înfășurătoare se comportă așadar ca o stivă, fiecare punct dat ajungând acolo o dată (după ce eventual a eliminat dintre punctele puse în stivă anterior). Timpul de rulare a codului descris mai sus este așadar de ordinul numărului de puncte care se dau.

Pentru a obține număr maxim de puncte de pe înfășurătoare nu vom scoate din stivă punctul din vârf atunci când D este 0 (la testarea unghiului).

Utimul lucru de care trebuie ținut cont este următorul: În cazul în care două puncte sunt coliniare cu originea în ce ordine le sortăm ? Observăm că ultimele puncte trebuie parcurse în ordine descrescătoare a distanței față de origine (pentru a rămâne pe înfășurătoare număr maxim de puncte). Singurele puncte care trebuie parcurse în ordine crescătoare ale distanței față de origine sunt cele care fac unghiul minim cu axa ox (cele aflate pe dreapta determinată de punctele 0 și 1). Deci, ca al doilea criteriu de sortare vom considera descrescător după distanța față de origine, iar secvența de puncte de la început de pe dreapta determinată de 0 și 1 o vom simetriza.

Sursa

```
#include <fstream>
#include <algorithm>
#include <cmath>
#define INF 1002
using namespace std;
ifstream fin ("infasuratoareconvexa.in");
ofstream fout("infasuratoareconvexa.out");
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int cmp(const pair<int, int> &a, const pair<int, int> &b) {
    int d = det(0, 0, a.first, a.second, b.first, b.second);
    if (d != 0)
        return d > 0;
    else
        return a.first*a.first + a.second*a.second > b.first*b.first + b.second*b.second;
}
pair<int, int> v[103], s[103], aux;
int n, i, j, k, pminim;
int main() {
    fin>>n;
    pminim = 0;
    v[0].first = v[0].second = INF;
    for (i=1;i<=n;i++) {
        fin>>v[i].first>>v[i].second;
        if (v[i].second < v[pminim].second || ( (v[i].second ==
v[pminim].second)&&(v[i].first < v[pminim].first) ))
            pminim = i;
    }
    v[0] = v[pminim];
    v[pminim] = v[1];
    v[1] = v[0];
    for (i=1;i<=n;i++) {
        v[i].first -= v[0].first;
        v[i].second -= v[0].second;
    }
    sort(v+2, v+n+1, cmp);
    for (j=3;j<=n;j++)
        if (det(v[1].first, v[1].second, v[2].first, v[2].second, v[j].first,
v[j].second)) {
```

```
            break;
        }
    i=2;
    j--;
    while (i<j) {
        aux = v[i];
        v[i] = v[j];
        v[j] = aux;
        i++;
        j--;
    }
    s[1] = v[1];
    s[2] = v[2];
    k = 2;
    for (i=3;i<=n;i++) {
        while (k>=2&&det(s[k-1].first,s[k-1].second,s[k].first,s[k].second,v[i].first,
v[i].second)<0) {
            k--;
        }
        s[++k] = v[i];
    }
    fout<<k<<"\n";
    for (i=1;i<=k;i++)
        fout<<s[i].first + v[0].first<<" "<<s[i].second + v[0].second<<"\n";
    return 0;
}
```

8. Verificarea dacă un punct este în interiorul sau pe perimetrul unui poligon.

Enunț

Cerința

Se dau coordonatele în plan pentru n puncte care determină un poligon. Se mai dau coordonatele altor m puncte. Să se verifice, pentru fiecare dintre cele m puncte, dacă se găsește sau nu în interiorul (sau pe marginea) poligonului.

Date de intrare

Fișierul punctinpoligonsimplu.in conține pe prima linie două numere separate prin spațiu: `n` și `m`, reprezentând respectiv, numărul de vârfuri ale poligonului și numărul de puncte de testat. Pe următoarele `n` linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui vârf al poligonului. Acestea sunt date într-un sens de parcurgere a laturilor poligonului. Pe următoarele `m` linii se găsesc câte două numere separate printr-un spațiu, reprezentând abscisa respectiv ordonata câte unui punct care trebuie testat.

Date de ieșire

Fișierul punctinpoligonsimplu.out conține `m` linii și pe fiecare dintre ele sa află mesajul `DA` sau `NU` după cum punctul corespunzător este sau nu în interiorul poligonului.

Restricții și precizări

$1 \leq n,m \leq 100$

Numerele din fișierul de intrare sunt întregi cuprinse între -1001 și 1001

Poligonul nu este neapărat convex dar nu se autointersectează.

În primele două teste poligonul este convex.

Exemplu

punctinpoligonsimplu.in

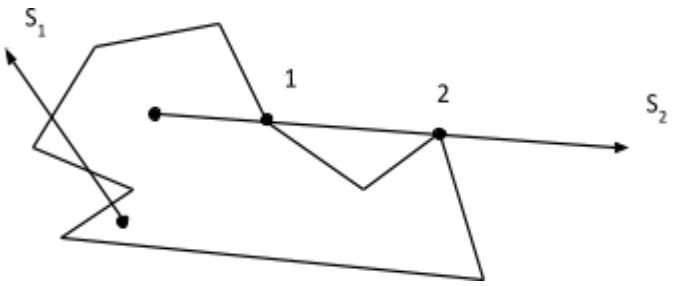
```
4 2
0 0
2 0
2 2
```

```
0 2
1 1
10 -2
punctinpoligonsimplu.out
DA
NU
```

Descrierea soluției

Dacă poligonul este convex testul pentru un punct poate fi făcut astfel: Calculăm semnul valorilor D pentru toate tripletele formate din punctul de testat și oricare două puncte vecine pe poligon (respectând mereu un sens de parcurgere). Dacă se obține întotdeanuna același semn atunci punctul se află în poligon.

Dacă poligonul nu este convex, observația cheie este următoarea: punctul este în poligon dacă și numai dacă există o semidreaptă (de fapt orice semidreaptă) cu originea în punctul de testat care va înțepa poligonul de un număr impar de ori. Astfel, testul ar avea drept cărămidă testarea intersecției a două segmente. În funcție de modul de alegere a semidreptei, pot apărea cazuri particulare (când semidreapta trece printr-un vârf al poligonului).



Observăm că o semidreaptă aleasă precum S_1 este una potrivită. O semidreaptă aleasă precum S_2 poate face să apară cazuri particulare mai dificil de tratat. De exemplu, dacă pe ea se află un punct ca 1, se observă că la întâlnirea lui trece din interiorul în exteriorul poligonului, iar dacă pe ea se află un punct ca 2, va rămâne în exterior (sau, după caz, în interior).

Pentru problema enuntata, datele fiind mici (atat numarul de puncte cât și coordonatele), este probabilitate foarte mare ca alegând aleator un punct la coordonate mai mari, dreapta ce se obține unindu-l cu punctul de testat să evite toate vârfurile poligonului. Așadar, în acest mod alegem semidreapta (generam aleator celălalt punct de pe dreapta suport până când această dreaptă nu trece prin niciun vârf al poligonului dat).

Sursa

```
#include <fstream>
#include <cmath>
#include <iomanip>
#include <stdlib.h>
#include <time.h>
using namespace std;
ifstream fin ("punctinpoligonsimplu.in");
ofstream fout("punctinpoligonsimplu.out");
int det(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    return (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
}
int punctPeSegment(int x1, int y1, int x2, int y2, int x3, int y3) {
    int d = det(x1, y1, x2, y2, x3, y3);
    if (d!=0)
        return 0;
```

```
    if (x1 == x3 && y1 == y3)
        return 1;
    if (x2 == x3 && y2 == y3)
        return 1;
    if ((x3-x1) * (x3-x2) < 0 || (y3-y1) * (y3-y2) < 0)
        return 1;
    else
        return 0;
}
int n, m, i, j, X1, Y1, intersect, testcurent, X3, Y3, X4, Y4, d1, d2, d3, d4;
pair<int, int> p[102];
int main() {
    fin>>n>>m;
    for (i=1;i<=n;i++)
        fin>>p[i].first>>p[i].second;
    p[0] = p[n];
    int X2 = 1002;
    int Y2 = 0;
    srand(time(0));
    for (j=1;j<=m;j++) {
        fin>>X1>>Y1;
        int ok = 0;
        for (i=0;i<n;i++) {
            if (punctPeSegment(p[i].first, p[i].second, p[i+1].first, p[i+1].second,
                                                                        X1, Y1)) {
                ok = 1;
                break;
            }
        }
        if (ok) {
            fout<<"DA\n";
            continue;
        }
        do {
            X2 = 1002 + rand()%1000;
            Y2 = 1002 + rand()%1000;
            intersect = 0;
            testcurent = 1;
            for (i=0;i<n;i++) {
                X3 = p[i].first;
                Y3 = p[i].second;
                X4 = p[i+1].first;
                Y4 = p[i+1].second;

                d1 = det(X3, Y3, X4, Y4, X1, Y1);
                d2 = det(X3, Y3, X4, Y4, X2, Y2);
                d3 = det(X1, Y1, X2, Y2, X3, Y3);
                d4 = det(X1, Y1, X2, Y2, X4, Y4);
                if (d1 == 0 && d2 == 0) {
                    testcurent = 0;
                    break;
                }
                if (punctPeSegment(X1, Y1, X2, Y2, X3, Y3) || punctPeSegment(X1, Y1,
                                                                    X2, Y2, X4, Y4)) {
                    testcurent = 0;
                    break;
                }
                if (d1 * d2 < 0 && d3 * d4 < 0)
                    intersect++;
            }
            if (testcurent) {
                if (intersect % 2 == 1) {
                    fout<<"DA\n";
                } else {
                    fout<<"NU\n";
```

```
                }
                break;
            }
        } while (1);
    }
    return 0;
}
```

8. Problema "Popândăi 2" (infoarena.ro)

Popândăii de pe "tarlaua veselă" au scăpat de atacul vulturilor și acum trebuie să se adăpostească de lupi în vizuinile lor. Aceste vizuini se pot identifica prin puncte avand coordonate întregi în plan și sunt dispuse în colțurile unui poligon convex. Pentru a fi protejați de atacul lupilor, popândăii vor să stabilească un perimetru de siguranță cât mai mare posibil, unde se pot mișca în voie. Acest perimetru va fi în formă de patrulater și va avea vârfurile situate în patru din cele `N` puncte care reprezintă vizuinile.

Cerința
Ajutați popândăii să determine zona de arie maximă care satisface condițiile de mai sus!

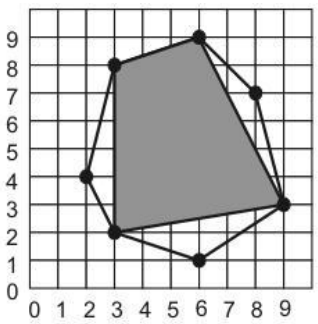
Date de Intrare
Pe prima linie a fișierului de intrare `popandai2.in` se află un număr întreg `N`, reprezentând numărul vizuinilor. Următoarele `N` linii conțin fiecare câte două numere întregi (X_i , Y_i) (separate printr-un spațiu) reprezentând coordonatele celei de-a i-a vizuini. Aceste coordonate vor fi date în ordine trigonometrică.

Date de Ieșire
Pe prima linie a fișierului de ieșire `popandai2.out` va fi afișat un singur număr real cu o zecimală exacta reprezentând aria maximă a patrulaterului căutat.

Restricții și precizări
- $4 \leq N \leq 1.000$
- $1 \leq X_i, Y_i \leq 30.000$
- Pentru `60%` din punctaj rezolvați problema pe cazul $3 \leq N \leq 300$

Exemplu

```
popandai2.in
7
3 2
6 1
9 3
8 7
6 9
3 8
2 4
popandai2.out
28.5
```



Descrierea soluției

Pe scurt, problema spune așa: dat fiind un poligon convex, se cere să determinăm un patrulater convex cu vârfurile dintre cele ale poligonului și de arie maximă.

Soluția 1.

Vom folosi `4` foruri, unul în altul, fixând în toate modurile cele `4` vârfuri ale patrulaterului. Timpul de calcul este de ordin n^4.

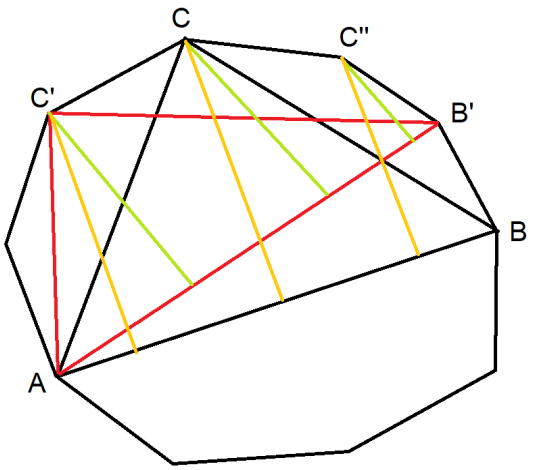
Soluția 2.

Observăm că dacă fixăm o diagonală a poligonului, de o parte și de alta a ei se formează două triunghiuri. Ne interesează să maximizăm aria fiecăruia dintre ele. Întrucât ele au baza stabilită (diagonala fixată), rămâne să alegem, pentru fiecare, celălalt vârf încât înălțimea să fie cât mai mare. Practic, avem un `for` de o parte a diagonalei fixate, pentru a alege punctul care împreună cu cele de pe diagonală maximizează aria triunghiului și încă un `for`, separat, pentru a face raționamentul același de cealălaltă parte a diagonalei. Timpul total de calcul ajunge la n^3 (avem nevoie de n^2 pentru a fixa diagonala).

Soluția 3.

Vom folosi ideea de la soluția anterioară, fixând o diagonală. Cu un `for` stabilim un punct al poligonului și cu al doilea `for` punctul diagonal opus. Să ne imaginam că suntem la o anume diagonală și, folosind rationamentul de la soluția anterioară, am determinat și cele două puncte care maximizează aria poligonului (adică aria fiecăruia dintre triunghiurile aflate de o parte și de alta a diagonalei).

Ne gândim acum că trecem la următoarea iterare a celui de-al doilea `for` (adică, alegem altă diagonală, păstrând pentru ea același punct de "început" ca și la anterioara, dar ca punct de final trecem la următorul de pe poligon). Acum este observația cheie: punctele care maximizează ariile celor două triunghiuri (pentru noua diagonală aleasă) sunt "după" (în sensul de parcurgere - să zicem trigonometric) față de cele de la poziția anterioară a diagonalei. Să analizăm figura următoare:



Considerăm diagonala AB. De o parte a ei, maximixăm aria alegând punctul C. Nu am mai figurat ce se întâmplă de cealaltă parte pentru că se face un raționament similar.

Dacă trecem la altă poziție a diagonalei (păstrăm pe A și ne mutăm cu celălalt capăt în B'), celălalt vârf al triunghiului fie rămâne C, fie va fi un punct de după el (de exemplu C'), dar în niciun caz un punct dinaintea lui (precum C'').

Iată o justificare rapidă: Întrucât C a fost cel mai depărtat punct de AB, segmentul portocaliu care îl proiectează pe C pe AB este mai lung decât cel portocaliu care îl proiectează pe C' pe AB. Mutându-ne cu diagonala către C' (adică A rămâne nemodificat și B "merge" în B'), proiecția verde a lui C pe AB' va fi și ea mai mare decât cea verde din C'' pe AB' (pentru o justificare mai riguroasă, observăm că se formează trapeze dreptunghice iar proiecția din C este tot timpul latura "paralelă" mai mare).

Așadar, mutând într-un sens extremitatea a doua a diagonalei, pentru determinarea vârfurilor pentru care se maximizează ariile în cele două triunghiutri, continuăm în același sens (sau rămânem pe loc!) pornind din vârfurile determinate la diagonala anterioară.

Astfel, avem nevoie de un for prin care fixăm extremitatea inițială a diagonalei, iar pentru el facem încă o parcurgere pentru a fixa cealaltă extremitate a diagonalei, și odată cu ea, și celelalte două parcurgeri pentru a alege optim celelalte vârfuri ale celor două triunghiuri (folosind, cum am justificat, faptul că se continuă în același sens de parcurgere din pozițiile de la pasul anterior). Timpul de executare este de ordin n^2.

Mai jos este o sursă pe această idee.

```
#include <fstream>
#define DIM 30010
#define x first
#define y second

using namespace std;

int n, i, j, nexti, nextj, S, maxim;

int Next(int i) {
    if (i < n)
        return i+1;
    else
        return 1;
}

int aria(int X1, int Y1, int X2, int Y2, int X3, int Y3) {
    int r = (X2-X1)*(Y3-Y1) - (X3-X1)*(Y2-Y1);
    if (r > 0)
        return r;
    else
        return -r;
}

pair<int, int> v[DIM];

int main () {
    ifstream fin ("popandai2.in");
    ofstream fout("popandai2.out");
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i].x>>v[i].y;

    for (int i=1;i<n;i++) {
        nexti = i;
        nextj = i+1;
        for (int j=i+1;j<=n;j++) {
            while (nexti != j &&
              aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nexti].x, v[nexti].y)<=
              aria(v[i].x, v[i].y, v[j].x, v[j].y, v[Next(nexti)].x, v[Next(nexti)].y)){
                nexti = Next(nexti);
            }
            while (nextj != i &&
```

```
              aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nextj].x, v[nextj].y)<=
              aria(v[i].x, v[i].y, v[j].x, v[j].y, v[Next(nextj)].x, v[Next(nextj)].y)){
                  nextj = Next(nextj);
              }

              S = aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nexti].x, v[nexti].y) +
                  aria(v[i].x, v[i].y, v[j].x, v[j].y, v[nextj].x, v[nextj].y);

              if (S > maxim)
                  maxim = S;
          }
      }

      fout<<maxim/2;
      if (maxim%2 == 0)
          fout<<".0\n";
      else
          fout<<".5\n";
      return 0;
}
```

Ca și detalii de implementare:

- remarcăm folosirea funcției `Next` care ne ușurează trecerea la vârful următor, având nevoie să parcurgem circular și uneori fiind nevoie să trecem de la vârful `n` la vârful `1`.
- întrucât aria este fie număr întreg fie jumătate de întreg, lucrăm pe parcurs cu dublul valorilor (evitând deci numerele reale), iar la final afișăm `.0` sau `.5` în funcție de paritatea rezultatului.

9. Baleiere

În foarte multe cazuri prin baleiere, în contextul problemelor de geometrie, înțelegem faptul că avem o dreaptă imaginară, de exemplu verticală, care traversează planul de la -∞ la +∞, iar la întâlnirea diverselor elemente considerăm că apar evenimente pe care le tratăm.

Considerăm următoarea problemă:

Intersecție segmente

Cerința

Se dau `N` segmente în plan, fiecare fiind paralel cu una dintre axele de coordonate.
Determinați numărul total de puncte de intersecție între două segmente.

Date de intrare

Prima linie a fisierului `is.in` conține un număr `N`, ce reprezintă numărul de segmente. Fiecare din următoarele `N` linii conțin câte `4` numere, separate prin câte un spațiu: `X1 Y1 X2 Y2`. `X1` și `Y1` reprezintă abscisa respectiv ordonata unui capăt al segmentului iar `X2` și `Y2` abscisa respectiv ordonata celuilalt capăt.

Date de ieșire

Fișierul `is.out` conține un singur număr, pe prima linie, reprezentând valoarea cerută.

Restricții și precizări

- `1 ≤ N ≤ 100.000`
- `0 ≤ X1, Y1, X2, Y2 ≤ 300.000`

- oricare două puncte date sunt distincte
- nu există capete ale vreunui segment aflate pe alt segment

```
is.in
5
1 1 10 1
1 4 5 4
2 0 2 20
3 0 3 3
15 15 20 15

is.out
3
```

Descrierea soluției

O primă variantă de rezolvare este să analizăm segmentele "fiecare cu fiecare" și, atunci când întâlnim un segment orizontal testat cu unul vertical, verificăm dacă se intersectează (cu timp de calcul de ordin constant). Această soluție are însă timp de calcul de ordin n^2 și nu se va încadra în timp pentru seturile de date mari.

Să vedem acum o soluție mai bună.

Pentru fiecare punct care apare ca și capăt al vreunui segment reținem segmentul pe care apare și tipul acestui segment (orizontal sau vertical). Sortăm aceste puncte crescător după x (pe implementarea de mai jos, am introdus punctele într-o structură set, care le clasifică după x).

Analizăm punctele în ordine crescătoare a abscisei lor.

Când întâlnim un punct care se află pe un segment vertical, ne imaginăm lucrurile astfel:

- segmentul corespunzător punctului este dreapta verticală de baleiere (care se mișcă de la stâng la dreapta, întrucât punctele sunt sortate crescător după x);
- în acest moment sunt unele segmente orizontale la care li s-a întâlnit anterior doar capătul stâng;
- trebuie să vedem care dintre aceste segmente orizontale sunt intersectate de segmentul nostru vertical;

Avem trei tipuri de evenimente:

1. Întâlnim capătul stâng al unui segment orizontal. În acest moment, într-un vector de frecvență F, mărim cu 1 pe poziția valorii y a segmentului orizontal (semnificație: începe un segment orizontal, cele verticale întâlnite până la capătul său drept se poate să îl intersecteze).
2. Întâlnim capătul drept al unui segment orizontal. În acest moment, într-un vector de frecvență F, scădem cu 1 pe poziția valorii y a segmentului orizontal (semnificație: se termină un segment orizontal, deci cele verticale ce vor mai apărea nu îl mai pot intersecta).
3. Întâlnim un segment vertical. Practic, acum trebuie să vedem care este suma valorilor din F aflate între indicii dați de valorile y ale celor două capete ale segmentului vertical.

Următorul pas este să observăm că operațiile pe vectorul F le putem simula pe un arbore de intervale (sau chiar arbore indexat binar).

Astfel, dacă întâlnim un punct (x, y) capăt stâng al unui segment orizontal (evenimentul 1 de mai sus), facem update în `AINT` la poziția y (mărim cu 1).

Dacă întâlnim un punct (x, y) capăt drept al unui segment orizontal (evenimentul 2 de mai sus), facem update în AINT la poziția y (scădem cu 1).

Dacă întîlnim un punct (x1, y1) care se află pe un segment vertical ce are al doilea punct în (x1, y2), este suficient să facem query de sumă în AINT pentru intervalul (y1, y2). Așa cum spuneam, asta contorizează segmentele "începute" și "neterminate" (deci cu x de început în stânga x-ului curent și cu x de final în dreapta x-ului curent - întrucât, noi știm că analizăm punctele, crescător după x), și care au y cuprins între valorile y ale capetelor segmentului vertical. Valoarea obținută este chiar numărul de segmente intersectate de cel curent.

În sursa de mai jos remarcăm și un detaliu de implementare care ușurează lucrurile: pentru segmentele verticale luam în calcul doar unul dintre puncte ca generator de evenimente.

Am putut folosi arborele de intervale întrucât valorile y sunt relativ mici (cel mult 300000, deci pot reprezenta indicii elementelor unui vector). Daca valorile erau mai mari, întrucât numărul lor este ralativ mic (cel mult 100000), am fi putut folosi arborele de intervale după ce procedam în prealabil la o normalizare.

Timpul de calcul este de ordin n log n (avem pe de o parte sortarea cu această complexitate, apoi, la fiecare punct întâlnit facem o operație în arborele de intervale, având și ea timp de calcul de ordin log).

```
#include <fstream>
#include <set>
#define DIM 100010

using namespace std;

ifstream fin ("is.in");
ofstream fout("is.out");

struct segment {
    int x1;
    int y1;
    int x2;
    int y2;

    int getType() {
        if (y1 == y2)
            return 1;
        else
            return 2;
    }
};

set <pair <int, int> > s;

segment v[DIM];
int A[DIM*3*4*2];
int n, N, sol;

void update(int nod, int st, int dr, int poz, int val) {
    if (st == dr) {
        A[nod]+=val;
    } else {
        int mid = (st + dr)/2;
        if (poz <= mid)
            update(2*nod, st, mid, poz, val);
        if (poz > mid)
            update(2*nod+1, mid+1, dr, poz, val);
```

```
        A[nod] = A[2*nod] + A[2*nod+1];
    }
}

void query(int nod, int st, int dr, int a, int b) {
    if (a <= st && dr <= b) {
        sol += A[nod];
    } else {
        int mid = (st + dr)/2;
        if (a <= mid)
            query(2*nod, st, mid, a, b);
        if (b > mid)
            query(2*nod + 1, mid+1, dr, a, b);

    }
}

int main () {
    fin>>n;
    for (int i=1;i<=n;i++) {
        fin>>v[i].x1>>v[i].y1>>v[i].x2>>v[i].y2;
        if (v[i].x1 == v[i].x2) {
            if (v[i].y1 > v[i].y2)
                swap(v[i].y1, v[i].y2);
            s.insert(make_pair(v[i].x1, i));
            N = max(N, v[i].y1);
            N = max(N, v[i].y2);
        } else {
            if (v[i].x1 > v[i].x2)
                swap(v[i].x1, v[i].x2);
            s.insert(make_pair(v[i].x1, i));
            s.insert(make_pair(v[i].x2, i));
            N = max(N, v[i].y1);
            N = max(N, v[i].y2);
        }
    }

    for (set<pair<int, int> >::iterator it = s.begin(); it != s.end(); it++) {
        int x = it->first;
        int i = it->second;

        if (v[i].getType() == 2) {
            query(1, 0, N, v[i].y1, v[i].y2);
        } else {
            if (v[i].x1 == x)
                update(1, 0, N, v[i].y1, 1);
            else
                update(1, 0, N, v[i].y1, -1);
        }
    }
    fout<<sol;
    return 0;
}
```