Geometrie computațională I

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2016/2017

Plan

Ce este geometria computațională

- obiectele geometrice - punctele, liniile, poligoanele, etc. - constituie baza multor aplicații
- apariția disciplinei: 1975, Shamos
- algoritmii operează cu sau sunt ghidați de o serie de structuri de date caracteristice geometriei computaționale
- acestea includ aranjamente de obiecte geometrice, localizări, înfășuratoarea convexă, diagrame Voronoi, triangularizări

Aplicații

- grafică (”computer vision”, reconstruirea de imagini)
- robotică (mișcare în plan, vizibilitate)
- proiectare asistată de calculator (CAD)
- siteme informatice geografice (GIS)
- statistică

Exemplu: înfășuratoarea convexă

Dată o mulțime finită S de puncte în plan, să se determine cea mai mică mulțime convexă care include S.

Exemplu: intersecția de poligoane

Date două suprafețe poligonale, să se calculeze intersecția lor.

Exemplu: problema galeriei de arta

Podeaua unei galerii de arta este sub forma unui poligon. Se pune problema determinării numărului de paznici care să supravegheze complet întreaga galerie. Unghiul de vedere al unui paznic este de 360 de grade dar el nu poate vedea prin ziduri.

Formulare echivalentă: câte becuri sunt necesare pentru luminarea galeriei (se presupune că tavanul are aceeași formă ca podeaua și pereții nu reflectă lumina).

Variante:
– număr minim de paznici,
– numărul de paznici necesari pentru orice poligon cu n vârfuri, pentru un n dat

Vizual - simplu, Algoritmic - mai complicat

Instance: Se consideră un triunghi T și un punct P.
Question: Este P interior lui T?
Aceeași problemă dar cu un poligon convex PLG în loc de triunghiul T.

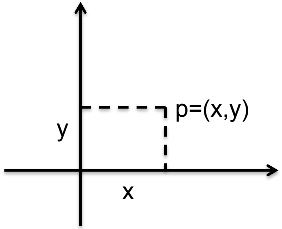
- dacă desenăm triunghiul (poligonul) și punctul, răspunsul este identificat vizual imediat;
- algoritmic nu e așa de simplu:
  – structuri de date pentru T (PLG) și P
  – formularea matematică a răspunsului
  – translatarea formulării matematice în limbaj algoritmic, utilizând termenii structurilor de date

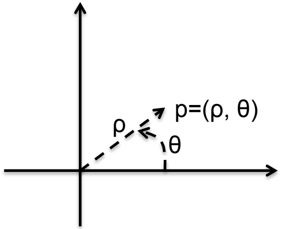
Plan

Plan

Puncte

Un punct P poate fi reprezentat prin coordonate carteziene, $P = (x, y)$, sau coordonate polare, $P = (\rho, \theta)$.





Tipuri de date pentru puncte

- structuri

Coordonate carteziene:

```
P |-> {x |-> 4, y |-> 3}
```

$\mathtt{P} \mapsto v_P,\ v_P \in \mathit{CPoint}$
$\mathit{CPoint} = \mathit{Str}\langle \mathtt{x} : \mathit{Float}, \mathtt{y} : \mathit{Float}\rangle = \{\{\mathtt{x} \to v_x\ \mathtt{y} \to v_y\} \mid v_x, v_y \in \mathit{Float}\}\}$

Coordonate polare:

```
P |-> {rho |-> 5, theta |-> 0.643501109}
```

$\mathtt{P} \mapsto v_P,\ v_P \in \mathit{PPoint}$
$\mathit{PPoint} = \mathit{Str}\langle \mathtt{rho} : \mathit{Float}, \mathtt{theta} : \mathit{Float}\rangle = \{\{\mathtt{rho} \to \rho\ \mathtt{theta} \to \theta\} \mid \rho, \theta \in \mathit{Float}\}\}$

Conversie polare $\mapsto$ carteziene

$(\rho, \theta) \mapsto (\rho \cdot \cos(\theta), \rho \cdot \sin(\theta))$ (timp uniform: $O(1)$)

```
cart(PP) {
  CP.x = PP.rho * cos(PP.theta);
  CP.y = PP.rho * sin(PP.theta);
  return CP;
}

pi = 3.14159265359;

PP1.rho = sqrt(2);
PP1.theta = pi / 4;
CP1 = cart(PP1);
```

```
$ alki polar2cart.alk
State:

   CP1 |-> { (x -> 9.9999999999994837e-01) (y -> 1.0000000000000517e+00) }
   PP1 |-> { (rho -> 1.4142135623730951e+00)
             (theta -> 7.8539816339750002e-01) }
   pi |-> 3.1415926535900001e+00
```

Conversie carteziene $\mapsto$ polare 1/6

La o primă privire:

$$\rho = \sqrt{x^2 + y^2}$$

- $x = 0$

$$\theta' = \begin{cases} \frac{\pi}{2} & , y > 0 \\ -\frac{\pi}{2} & , y < 0 \\ \text{nedefinit} & , y = 0 \end{cases}$$

- $x \neq 0$

$$\theta' = \mathrm{atan}(\frac{y}{x})$$

$$\theta = \begin{cases} \theta' & , \theta' \geq 0 \\ \theta' + 2\pi & , \theta' < 0 \end{cases}$$

Convenim $\theta' = 0$ dacă $x = y = 0$.

Conversie carteziene $\mapsto$ polare 2/6

Relațiile din slide-ul precedent le descriem ca un algorithm:

```
polar(CP) {
  PP.rho = sqrt(CP.x * CP.x + CP.y * CP.y);
  if (CP.x == 0.0) {
    if (CP.y < 0.0) theta1 =  0.0 - pi/2.0;
    else theta1 =  pi/2.0;
  }
  else {
    theta1 = atan(CP.y / CP.x);
  }
  if (theta1 >= 0) PP.theta = theta1;
  else PP.theta = theta1 + 2 * pi;
  return PP;
}
```

Conversie carteziene $\mapsto$ polare 3/6

Un prim test:

```
CP1  = { x -> 1.0 y -> 1.0};
PP1 = polar(CP1);
```

```
$ alki cart2polar.alk
State:

    CP1 |-> { (x -> 1.0000000000000002e+00) (y -> 1e+00) }
    PP1 |-> { (rho -> 1.414213562373095le+00)
                (theta -> 7.853981633974483e-01) }
```

Intermezzo: atenție la erorile de calcul!!

$P == cartezian(polar(P))$

```
CP1  = { x -> 1.0 y -> 1.0};
PP1 = polar(CP1);
CP11 = cart(PP1);

if (CP1.x == CP11.x) b = true;
else b = false;

$ alki cart2polar.alk
State:

    ...
    CP1  |-> { (x -> 1e+00) (y -> 1e+00) }
    CP11 |-> { (x -> 1.0000000000000002e+00) (y -> 1e+00) }
    b    |-> false
```

Conversie carteziene ↦ polare 4/6

Al doilea test:

```
CP2  = { x -> -1.0 y -> 1.0};
PP2 = polar(CP2);
CP2 = cart(PP2);

$ alki cart2polar.alk
State:

    CP2 |-> { (x -> 1.000000000004137e+00)
              (y -> -9.999999999958644e-01) }
    PP2 |-> { (rho -> 1.414213562373095le+00)
              (theta -> 5.497787143782518e+00) }
    ...
```

Oops! Ceva nu e in regulă . . . (ce?)

Conversie carteziene $\mapsto$ polare 5/6

$$\mathrm{atan} : \mathbb{R} \to (-\frac{\pi}{2}, \frac{\pi}{2})$$

Trebuie determinat cadranul la care aparține punctul:

$$\theta' = \begin{cases} \mathrm{atan}(\frac{y}{x}) & , x > 0 \\ \mathrm{atan}(\frac{y}{x}) + \pi & , x < 0 \land y \geq 0 \\ \mathrm{atan}(\frac{y}{x}) - \pi & , x > 0 \land y < 0 \end{cases}$$

$$\theta = \begin{cases} \theta' & , \theta' \geq 0 \\ \theta' + 2\pi & , \theta' < 0 \end{cases}$$

În multe limbaje de programare θ' este notată cu `atan2()`.

Conversie carteziene $\mapsto$ polare 6/6

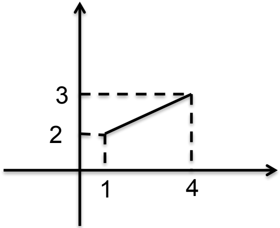
```
polar(CP) {
  PP.rho = sqrt(CP.x * CP.x + CP.y * CP.y);
  if (CP.x == 0.0) {
    if (CP.y < 0.0) theta1 =  0.0 - pi/2.0;
    else theta1 =  pi/2.0;
  } else {
    arctg = atan(CP.y / CP.x);
    if (CP.x >= 0.0) theta1 = arctg;
    else {
      if (CP.y < 0.0) theta1 =  arctg - pi;
      else theta1 = arctg + pi; }
  }
  if (theta1 >= 0) PP.theta = theta1;
  else PP.theta = theta1 + 2 * pi;
  return PP;
}
```

(timp uniform: $O(1)$)

Segmente

Un segment este reprezentat de o pereche de puncte:
`{A -> {x -> 1, y |-> 2} B -> {x -> 4, y -> 3}}`

Accesarea coordonatelor: `A.x` `A.y` `B.x` `B.y` ...



$$Segm = Str\langle \texttt{A} : Point, \texttt{B} : Point\rangle = \{\{\texttt{rho} \to \rho\ \texttt{theta} \to \theta\} \mid \rho, \theta \in Float\}$$

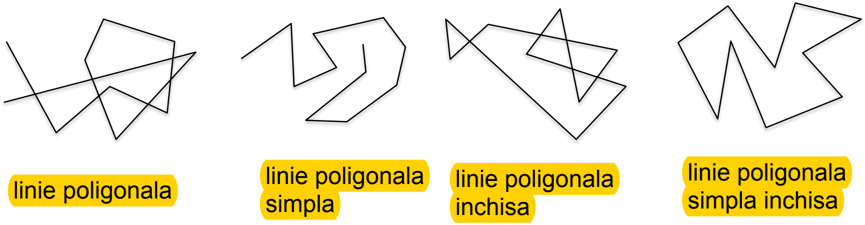
$$Point \in \{CPoint, PPoint\}$$

Linii poligonale

Structura de date: listă liniare de puncte

Pot fi:

- simple
- închise/deschise



Structuri de date pentru linii poligonale

- liste liniare

$PolygLine = List\langle Point\rangle$

Crearea unei liste cu punctele:

```
A = { x -> 3 y -> 5 };
B = { x -> 2 y -> 1 };
C = { x -> 0 y -> 0 };
```

se poate face prin

```
L = emptyList;
L.pushBack(A);
L.pushBack(B);
L.pushBack(C);
```

sau

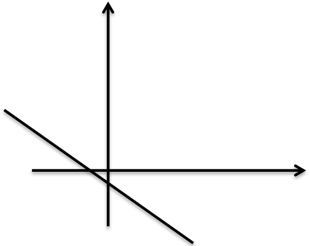
```
L = emptyList;
L[0] = A;
L[1] = B;
L[2] = C;
```

sau

```
L = emptyList;
L.insert(0, A);
L.insert(1, B);
L.insert(2, C);
```

Dreapta

O dreaptă este reprezentată printr-o ecuatie liniară: $a \cdot x + b \cdot y + c = 0$



Structura de date: structura
$Line = Str\langle \mathtt{a} : Float, \mathtt{b} : Float, \mathtt{c} : Float \rangle$

Exemplu: dreapta $3x + 4y + 2 = 0$ este reprezentată de structura

```
d |-> {a -> 3 b -> 4  c -> 2}
```

Dreapta care trece prin două puncte distincte P și Q 1/3

- se consideră sistemul:

$$\begin{cases} \mathtt{d.a} * \mathtt{P.x} + \mathtt{d.b} * \mathtt{P.y} + \mathtt{d.c} = 0 \\ \mathtt{d.a} * \mathtt{Q.x} + \mathtt{d.b} * \mathtt{Q.y} + \mathtt{d.c} = 0 \end{cases}$$

- îhm, două ecuații și trei necunoscute?

  Distingem cazurile:

- dreapta este paralelă cu Oy: rezultă $P.x = Q.x$ iar ecuația este $x = P.x$ (sau $x = Q.x$), i.e., $d.a = 1, d.b = 0, d.c = -P.x$

Dreapta care trece prin două puncte distincte P și Q 2/3

- dreapta NU este paralelă cu Oy: rezultă $d.b \neq 0$ și sistemul devine

  $$\begin{cases} \mathtt{P.y} = \mathtt{m} * \mathtt{P.x} + \mathtt{n} \\ \mathtt{Q.y} = \mathtt{m} * \mathtt{Q.x} + \mathtt{n} \end{cases}$$

  unde $m = -\frac{a}{b} = \frac{P.y - Q.y}{P.x - Q.x}$, $n = -\frac{c}{a} = P.y - \frac{P.y - Q.y}{P.x - Q.x} * P.x$
  Luăm $d.a = -m, d.b = 1, d.c = -n$ și obținem forma mult mai cunoscută $y = mx + n$ pentru ecuație, unde m este panta dreptei.

- $P = Q$: dreapta este nedefinită; luăm $d.a = d.b = d.c = 0$

Dreapta care trece prin două puncte distincte P și Q 3/3

Algoritmul final:

```
line(P,Q) {
  if (P.x == Q.x && P.y == Q.y)
    return {a -> 0.0 b -> 0.0 c -> 0.0};
  if (P.x == Q.x) {
    l.a = 1.0;
    l.b = 0.0;
    lc = P.x;
    return l;
  }
  l.a = 0.0 - (P.y - Q.y) / (P.x - Q.x);
  l.b = 1.0;
  l.c = 0.0 - P.y - l.a * P.x;
  return l;
}
```

Plan

Poziționarea unui punct față de o dreaptă

Fie `P` un punct și `d` o dreaptă. Relativ la d, P se poate afla:

- într-un semiplan: `d.a * P.x + d.b * P.y + c > 0`
- pe dreaptă: `d.a * P.x + d.b * P.y + c = 0`
- în celălalt semiplan: `d.a * P.x + d.b * P.y + c < 0`

Notăm cu (d, P) semiplanul determinat de dreapta d și punctul P.

Poziționarea față de un segment `AB`:

- se determină dreapta suport
- dacă se află pe dreaptă, se verifică dacă este între `A` și `B`

Se poate testa și în ordine inversă.

Timp uniform: $O(1)$

Intersecția a două drepte 1/2

$$\begin{cases} a_1 \cdot x + b_1 \cdot y + c_1 = 0 \\ a_2 \cdot x + b_2 \cdot y + c_2 = 0 \end{cases}$$

$$det = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \quad det_x = \begin{vmatrix} -c_1 & b_1 \\ -c_2 & b_2 \end{vmatrix} \quad det_y = \begin{vmatrix} a_1 & -c_1 \\ a_2 & -c_2 \end{vmatrix}$$

$det = 0$, drepte paralele (excludem cazul când coincid)

$det \neq 0$, sistemul are solutie unică: $x = \dfrac{det_x}{det}$, $y = \dfrac{det_y}{det}$

În notația structurilor de date pentru drepte:

$$det = \begin{vmatrix} \texttt{d1.a} & \texttt{d1.b} \\ \texttt{d2.a} & \texttt{d2.b} \end{vmatrix} = \texttt{d1.a} * \texttt{d2.b} - \texttt{d1.b} * \texttt{d2.a}$$

Intersecția a două drepte 2/2

```
lineIntersection(l1, l2) {
  det = l1.a *l2.b - l1.b*l2.a;
  detx = l1.b*l2.c - l1.c *l2.b;
  dety = l1.c*l2.a - l1.a *l2.c;
  if (det == 0.0) return emptySet;
  P.x = detx / det;
  P.y = dety / det;
  return singletonSet(P);
}
```

Timp uniform: $O(1)$

Intersecția a două segmente (când există)

Soluția 1:

- se determină punctul P de intersecție a dreptelor suport;
- se verifică dacă P aparține segmentelor;

Soluția 2:

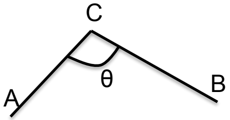
- se verifică dacă pentru fiecare segment capetele sale sunt deoparte și de alta a dreptei suport determinate de celălalt segment (detalii pe tablă);

Soluția 3 (sweep line):

- se baleiază (mătură) planul cu o dreaptă orizontală (sau verticală) și se analizează pozițiile punctelor eveniment (oarecum asemănător ca la 2);

Timp uniform: $O(1)$

Unghiul convex format de trei puncte 1/3



- distanța dintre două puncte:

$\mathtt{dist}(P, Q) = \sqrt{(Q.x - P.x) * (Q.x - P.x) + (Q.y - P.y) * (Q.y - P.y)}$

- se aplică teorema cosinusului

```
a = dist(C, B);
b = dist(C, A);
c = dist(A,B);
theta = acos((a*a + b*b - c*c)/ 2*a*b);
```

Timp uniform: $O(1)$

Unghiul convex format de trei puncte 2/3

```
dist(P, Q) {
  d1 = (Q.x-P.x)*(Q.x-P.x);
  d2 = (Q.y-P.y)*(Q.y-P.y);
  return sqrt(d1 + d2);
}

angle(A, C, B) {
  a = dist(C, B);
  b = dist(C, A);
  c = dist(A, B);
  return acos((a*a + b*b - c*c)/ (2*a*b));
}
```

Unghiul convex format de trei puncte 3/3

```
A = {x -> 1.0 y -> 1.0};
B = {x -> 3.0 y -> 1.0};
C = {x -> 2.0 y -> 2.0};
D = {x -> 5.0 y -> 1.0};
theta1 = angle(A, B, C);
theta2 = angle(B, C, A);
theta3 = angle(B, A, C);
theta4 = angle(A, B, D);

$ alki angle.alk
State:
    ...
    theta1 |-> 7.853981633974485e-01
    theta2 |-> 1.570796326794896 3e+00
    theta3 |-> 7.853981633974485e-01
    theta4 |-> 3.141592653589793 1e+00
```

Orientarea a trei puncte (primitiva `ccw`) 1/3

CCW = "Counter-Clock-Wise" (sensul invers arcelor de ceasornic)

Fie A, B, C trei puncte.

$$det(A, B, C) = \begin{vmatrix} \mathtt{A.x} & \mathtt{A.y} & 1 \\ \mathtt{B.x} & \mathtt{B.y} & 1 \\ \mathtt{C.x} & \mathtt{C.y} & 1 \end{vmatrix}$$

- $det(A, B, C) > 0$: A, B, C formează un ciclu în sens invers arcelor de ceasornic (întoarcere stânga)
- $det(A, B, C) < 0$: A, B, C formează un ciclu în sensul arcelor de ceasornic (întoarcere dreapta)
- $det(A, B, C) = 0$: A, B, C sunt coliniare

Timp uniform: $O(1)$

Convenție: $det(A, B, C) \rightarrow$ `sign2xTriArea(A,B,C)` (vom vedea mai târziu de ce)

Orientarea a trei puncte (primitiva ccw) 2/3

```
sign2xTriArea(A, B, C) {
  d1 = B.y * A.x + C.y * B.x + A.y * C.x;
  d2 = C.x * B.y + B.x * A.y + A.x * C.y;
  return d1 - d2;
}

ccw(A, B, C)
/*
  turn left = +1;
  turn right = -1;
  colinear = 0;
*/
{
  ax2 = sign2xTriArea(A, B, C);
  if (ax2 > 0.0) return 1;
  if (ax2 < 0.0) return -1;
  return 0;
}
```

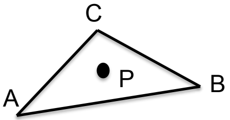
Orientarea a trei puncte (primitiva ccw) 3/3

```
A = {x -> 1.0 y -> 1.0};
B = {x -> 3.0 y -> 1.0};
C = {x -> 2.0 y -> 2.0};

t1 = ccw(A, B, C);
t2 = ccw(A, C, B);
t3 = ccw(A, B, B);
```

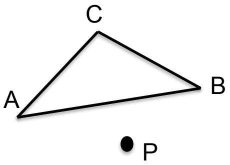
```
$ alki ccw.alk
State:

    A |-> { (x -> 1e+00) (y -> 1e+00) }
    B |-> { (x -> 3e+00) (y -> 1e+00) }
    C |-> { (x -> 2e+00) (y -> 2e+00) }
    t1 |-> 1
    t2 |-> -1
    t3 |-> 0
```

Plan

Localizarea unui punct față de un triunghi



`ccw(P, A, B)`, `ccw(P, B, C)` și `ccw(P, C, A)` au toate același semn.



`ccw(P, A, B)`, `ccw(P, B, C)` și `ccw(P, C, A)` NU au toate același semn.

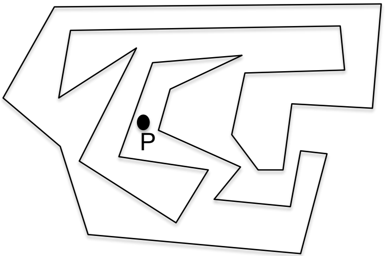
Timp uniform: $O(1)$

Localizarea unui punct față de o linie poligonală simplă închisă 1/6

Theorem (Jordan)

Orice curbă simplă închisă împarte planul în două regini distincte: interiorul liniei (mărginită) și exteriorul (nemărginită).

Instance: O linie poligonală simplă închisă L și un punct P.
Question: Aparține P interiorului lui L?

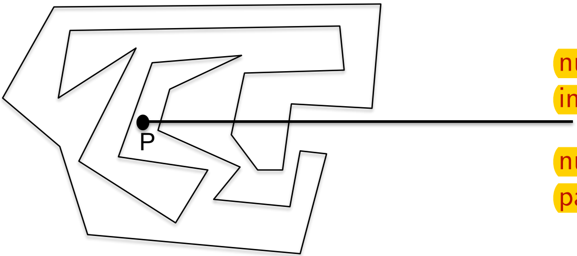


Localizarea unui punct față de o linie poligonală simplă închisă 1/6

Theorem (Jordan)

Orice curbă simplă închisă împarte planul în două regini distincte: interiorul liniei (mărginită) și exteriorul (nemărginită).

Instance: O linie poligonală simplă închisă L și un punct P.
Question: Aparține P interiorului lui L?

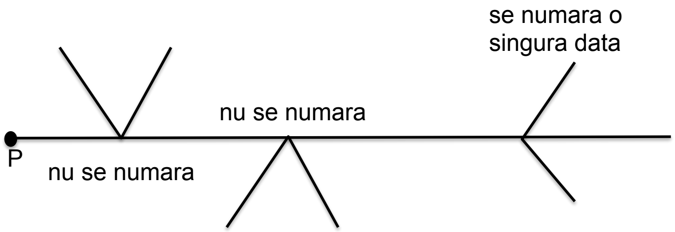


număr de intersecții impar ⇒ interior,

număr de intersecții par ⇒ exterior

Localizarea unui punct față de o linie poligonală simplă închisă 2/6

Număratul intersecțiilor trebuie făcut cu atenție deoarece sunt cazuri de excepție:



+ cazurile când o latură este inclusă în semidreaptă.

Calculul unei intersecții: $O(1)$
Determinarea numărului de intersecții: $O(n)$, n numărul de segmente ale liniei L

Localizarea unui punct față de o linie poligonală simplă închisă 3/6

Presupunem că $L[0]$ nu se află pe semidreaptă și că P nu se află pe linia poligonală L.

```
isInteriorOf(L, P) {
  L.pushBack(L[0]);
  counter = 0;
  for (i = 0; i < L.size()-1; i = j)
    counter = counter + count(i, j);
  return counter % 2 == 1;
}
```

Localizarea unui punct față de o linie poligonală simplă închisă 4/6

`count(i, out j)`:

- dacă $L[i]$ și $L[i+1]$ sunt de aceeași parte a semidreptei, atunci $\mathtt{j} = \mathtt{i} + 1$ și întoarce zero;
- altfel determină $\{Q\} = line(L[i], L[i+1]) \ \cap$ dreapta suport a semidreptei (Q există deoarece cazul când cele două drepte sunt paralele este exclus de itemul precedent);
- dacă $Q.x < P.x$, atunci $\mathtt{j} = \mathtt{i} + 1$ și întoarce zero;
- dacă $Q.x \geq P.x$ și $Q \neq L[i+1]$, atunci $\mathtt{j} = \mathtt{i} + 1$ și întoarce unu;
- dacă $Q.x \geq P.x$ și $Q = L[i+1]$:
  - determină primul $j > i + 1$ care nu aparține semidreptei (rezultă că nu aparține nici dreptei suport);
  - dacă $L[i]$ și $L[j]$ sunt de aceeași parte a semidreptei, atunci întoarce zero;
  - altfel întoarce unu;

Localizarea unui punct față de o linie poligonală simplă închisă 5/6

- dreapta suport a semidreptei:

```
dP.a = 0; dP.b = 1; dP.c =  0 - P.y;
```

- $L[i]$ și $L[i+1]$ ($L[j]$) sunt de aceeași parte a semidreptei:

```
onTheSameSide(P, Q, l) {
  return (l.a * P.x + l.b * P.y + l.c) *
         (l.a * Q.x + l.b * Q.y + l.c) > 0;
}
```

- $\{Q\} = line(L[i], L[i+1]) \cap$ dreapta suport a semidreptei:

```
Q = lineIntersection(line(L[i], L[i+1]), dP);
```

- determină primul $j > i+1$ care nu aparține semidreptei:

```
j = i+1;
while(L[j].y == P.y && L[j].x >= P.x) ++j;
```

Localizarea unui punct față de o linie poligonală simplă închisă 6/6

```
count(i, out j) {
  dP. a = 0; dP.b = 1; dP.c =  0 - P.y;  //dreapta suport a semidreptei
  if(onTheSameSide(L[i], L[i+1], dP)) {
    j = i+1;
    return 0;
  }
  Q = lineIntersection(line(L[i], L[i+1]), dP)[0];
  if (Q.x < P.x) {
    j = i+1;
    return 0;
  }
  if (Q.x != L[i+1].x || Q.y != L[i+1].y) {
    j = i+1;
    return 1;
  }
  j = i+1;
  while(L[j].y == P.y && L[j].x >= P.x) ++j;
  if(onTheSameSide(L[i], L[j], dP))
    return 0;
  return 1;
}
```

Plan

Diagonală (definiția)

O diagonală a unui poligon L este un segment AB determinat de două vârfuri A și B cu proprietatea că orice punct Q din AB, $Q \notin \{A, B\}$, se află în interiorul lui L.

Proposition

Orice poligon cu $n \geq 3$ vârfuri are cel puțin un vârf convex.

Hint

Cel mai de jos-dreapta.

Proposition

Orice poligon cu $n \geq 4$ vârfuri are cel puțin o diagonală.

Hint

Se utilizează propoziția de mai sus.

Cum determinăm o diagonală?

Definim o noțiune mau slabă, cea de candidat la diagonală:

- unește două vârfuri nealăturate ale poligonului
- nu intersectează alte laturi ale poligonului

Operații auxiliare:

- utilizăm iteratori pentru a referi vârfurile poligonului
- operație auxiliară: `isIntSeg(A, B, C, D)` - testează dacă segmentele AB și CD se intersectează

Candidat la diagonală: algoritm

```
/*
  tests if AB is a internal or external diagonal of the poligon L,
  where A = *a, B = *b, and a, b are iterators associated to L
*/
isDiagKind(a, b, L) {
  A = *a;  B = *b;
  for (p = L.first(); p != L.end(); ++ p) {
    q = p +% 1;
    if (( *p  == A && *q == B) ||
        ( *p  == B && *q == A))
      return false;
    if ( *p != A && *p != B &&
         *q != A && *q != B &&
         isIntSeg(A, B, *p, *q))
      return false;
  }
  return true;
}
```

Candidat la diagonală: test

```
test(out t) {
  L = < {x -> 1.0 y -> 1.0},
        {x -> 2.0 y -> 2.0},
        {x -> 3.0 y -> 1.0},
        {x -> 4.0 y -> 2.0},
        {x -> 5.0 y -> 1.0},
        {x -> 5.0 y -> 4.0},
        {x -> 3.0 y -> 5.0},
        {x -> 1.0 y -> 4.0} >;
  p0 = L.first();
  t[0] = isDiagKind(p0, p0+2, L);
  t[1] = isDiagKind(p0, p0+3, L);
}
t = [];
test(t);

$ alki isdiagkind.alk
State:

    t |-> [ true, false ]
```

Candidat la diagonală: intern sau extern

Un candidat la diagonală poate fi

- inclus în interiorul poligonului sau
- inclus în exteriorul poligonului (detalii pe tablă)

O diagonală este un candidat inclus în interiorul poligonului.

Un candidat AB se află în interior dacă B este interior unghiului din A și dacă A este interior unghiului din B.

Testul de diagonală: algoritm

```
/*
  tests if AB is a diagonal in L, where A = *a, B = *b
*/
isDiag(a, b, L) {
  return isDiagKind(a, b, L) &&
         inCone(a, *b) &&
         inCone(b, *a);
}
```

Test pentru interiorul unui unghi: definiția

Problema: test dacă A este interior vârfului P referit de p în poligonul L.

Se disting două cazuri:

- vârful P este convex: vecinii lui P se găsesc de o parte și de alta a lui PA
- vârful P este concav (reflex): se reduce la primul (A nu se află în unghiul convex din exterior)

Detalii pe tablă.

Test pentru interior unghiului: algoritm

```
/*
  tests if A is interior to the cone defined by
  the vertex *p of the poligon L
  (p is a iterator associated to L)
*/
inCone(p, A) {
  P = *p;
  Pprev = *(p -% 1);
  Pnext = *(p +% 1);
  if (ccw(P, Pnext, Pprev) == 1) // P is a convex vertex
    return ccw(P, A, Pprev) + ccw(A, P, Pnext) == 2;
  else // P is a concav (reflex) vertex
    return ccw(P, A, Pnext) + ccw(A, P, Pprev) != 0;
}
```

Test pentru interior unghiului: testare algoritm 1/2

```
test(out t) {
  A[0] = {x -> 1.0 y -> 1.0};
  A[1] = {x -> 2.0 y -> 2.0};
  A[2] = {x -> 3.0 y -> 1.0};
  A[3] = {x -> 4.0 y -> 2.0};
  A[4] = {x -> 5.0 y -> 1.0};
  A[5] = {x -> 5.0 y -> 4.0};
  A[6] = {x -> 3.0 y -> 5.0};
  A[7] = {x -> 1.0 y -> 4.0};
  L = emptyList;
  for (i = 0; i < A.size(); ++i)
    L.pushBack(A[i]);

  p = L.first();
  p = p + 1;  // *p is A[1]
  B1 = {x -> 2.0 y -> 0.0};
  t[0] = inCone(p, A[6]);
  t[1] = inCone(p, A[7]);
  t[2] = inCone(p, B1);
```

Test pentru interior unghiului: testare algoritm 2/2

```
  p = p + 1; // *p is A[2]
  t[3] = inCone(p, A[4]);
  t[4] = inCone(p, A[5]);
  t[4] = inCone(p, A[6]);
}

t = [];
test(t);

alki incone.alk
State:

    t |-> [ true, true, false, false, true ]
```

Plan

Triangularizare (definiția)

Theorem (Triangularizare)

Orice poligon cu $n \geq 3$ poate fi împărțit în triunghiuri ducând diagonale.

Demonstractia prin inducție după n.

Graful dual al unei triangularizări

Graful dual unei triangularizări:

- vârfuri: triunghiuri ale traingularizării
- există muchie între două vârfuri dacă acestea partajează o diagonală.

Proposition

Graful dual al unei triangularizări a unui poligon este un arbore.

"Ureche" a unui poligon

O ureche a unui poligon L este o secvență de trei vârfuri consecutive A, B, C a.î. AC este diagonală. B se numește vârful urechii.

Dacă A, B, C este ureche atunci $\widehat{ABC}$ este convex ($< \pi$).

Theorem

Orice poligon cu $n > 3$ vârfuri are cel puțin două urechi care nu se suprapun.

Urechile sunt frunze în arborele dual.

Algoritmul de triangularizare scris în termenii din domeniul problemei

- se determină statusul de "ureche" pentru fiecare vârf
- cât timp poligonul procesat are mai mult de 3 vârfuri
  - caută un vârf ureche
  - adaugă diagonala determinată de vârfurile vecine
  - elimină vârful ureche
  - actualizează statusul de "ureche"

Statusul de "ureche"

Un vârf este ureche dacă vecinii lui formează o diagonală.

```
/*
  set the ear status for each vertex in L
*/
setEarStatus(out L) {
  for (p = L.first(); p != L.end(); ++ p)
    *p.ear = isDiag(p -% 1, p +% 1, L);
}
```

Fiecare vârf are un câmp suplimentar `ear`, care va memora valoarea booleană ce caracterizează statusul de "ureche".

Caută un vârf ureche

```
for (p = L.first(); !(*p.ear); ++p) {}
```

Execuția se termină întotdeauna deoarece orice pligon cu $n \geq 3$ vârfuri are cel puțin o ureche.

Actualizarea statusului de "ureche"

- numai vecinii vârfului urechii își pot modifica statusul
  Detalii pe tablă

```
*(p -% 1).ear = isDiag(p -% 2, p +% 1, L);
*(p +% 1).ear = isDiag(p -% 1, p +% 2, L);
```

Triangularizare: algoritm

```
triangulate(L, out D) {
  D = emptySet;
  setEarStatus(L);
  while(L.size() > 3) {
    // search for an ear
    for (p = L.first(); !(*p.ear); ++p) {}
    // add the diagonal
    d.A.x = *(p -% 1).x;  d.A.y = *(p -% 1).y;
    d.B.x = *(p +% 1).x;  d.B.y = *(p +% 1).y;
    D.pushBack(d);
    // reset ear status for p-1 and p+1
    *(p -% 1).ear = isDiag(p -% 2, p +% 1, L);
    *(p +% 1).ear = isDiag(p -% 1, p +% 2, L);
    // remove *p
    p->delete();
  }
}
```

Triangularizare: testare

```
test(out D) {
  L = < {x -> 1.0 y -> 1.0},
        {x -> 2.0 y -> 2.0},
        {x -> 3.0 y -> 1.0},
        {x -> 4.0 y -> 2.0},
        {x -> 5.0 y -> 1.0},
        {x -> 5.0 y -> 4.0},
        {x -> 3.0 y -> 5.0},
        {x -> 1.0 y -> 4.0} >;
  triangulate(L, D);
}
test(D);

$ alki triangulate.alk
State:

  D |-> { { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 2e+00 y -> 2e+00 } },
          { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 3e+00 y -> 1e+00 } },
          { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 4e+00 y -> 2e+00 } },
          { A -> {x -> 4e+00 y -> 2e+00 } B -> {x -> 5e+00 y -> 4e+00 } },
          { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 5e+00 y -> 4e+00 } }
        }
```

Triangularizare: analiza 1/2

Corectitudine:

- invariant:
  - L este un poligon nediagonalizat cu vârfurile o submulțime a celui inițial
  - D este o triangularizare a porțiunii rămase prin eliminarea lui L din poligonul inițial

Triangularizare: analiza 2/2

Timp:

- dimensiune instanță: $n = L.size()$
- cazul cel mai nefavorbil:
  - `setEarStatus(L)`: $O(n^2)$
  - bucla `while`:
    - căutarea urechii: $O(n)$
    - `isDiag(...)`: $O(n)$
    - restul operațiilor: $O(1)$

  și se execută de $n - 3$ ori
- $\implies T(n) = O(n^2)$

Plan

Aria unui triunghi 1/2

Theorem

Dacă A, B, C formează o întoarcere stânga atunci `sign2xTriArea`(A, B, C) este dublul ariei triunghiului ABC (de unde si numele).

```
a1 = sign2xTriArea(A, B, C);
a2 = sign2xTriArea(A, C, B);

$ alki ccw.alk
State:

    A |-> { (x -> 1e+00) (y -> 1e+00) }
    B |-> { (x -> 3e+00) (y -> 1e+00) }
    C |-> { (x -> 2e+00) (y -> 2e+00) }
    a1 |-> 2e+00
    a2 |-> -2e+00
```

Aria unui triunghi 2/2

Theorem

Dacă A, B, C formează o întoarcere stânga și P un punct oarecare din plan, atunci `sign2xTriArea`$(P, A, B)+$ `sign2xTriArea`$(P, B, C)+$ `sign2xTriArea`(P, C, A) este dublul ariei triunghiului ABC.

```
P =  {x -> -7.0 y -> -9.0};

a3 = sign2xTriArea(P, A, B) +
     sign2xTriArea(P, B, C) +
     sign2xTriArea(P, C, A);

$ alki ccw.alk
STate:

    A |-> { (x -> 1e+00) (y -> 1e+00) }
    B |-> { (x -> 3e+00) (y -> 1e+00) }
    C |-> { (x -> 2e+00) (y -> 2e+00) }
    P |-> { (x -> -7e+00) (y -> -9e+00) }
    a3 |-> 2e+00
```

Aria unui poligon 1/2

Theorem

Dublul ariei unui poligon $P_0P_1 \ldots P_{n-1}$, $n \geq 3$, cu vârfurile parcurse in sens invers arcelor de ceasornic, este

$$\begin{aligned}&\texttt{sign2xTriArea}(Q, P_0, P_1) + \texttt{sign2xTriArea}(Q, P_1, P_2) + \cdots + \\ &\texttt{sign2xTriArea}(Q, P_{n-2}, P_{n-1}) + \texttt{sign2xTriArea}(Q, P_{n-1}, P_0)\end{aligned} \tag{1}$$

unde Q este un punct oarecare din plan.

Demonstrația prin inducție după n și se utilizează că poligonul are o ureche.

Aria unui poligon 2/2

Suma (1) este egală cu

$$\sum_{i=0}^{n-1}(P_i.xP_{i+1}.y - P_{i+1}.xP_i.y)$$

care este egală cu

$$\sum_{i=0}^{n-1}(P_i.x + P_{i+1}.x)(P_{i+1}.y - P_i.y)$$

considerând Q originea și făcând calcule.

Proposition

Aria unui poligon cu $n \geq 3$ vârfuri poate fi calculat cu n înmuțiri și $2n$ adunări/scăderi.