Aula 14 · Geometria 2D

Desafios de Programação

Fernando Kiotheka Vinícius Tikara Date

UFPR

21/06/2023

Geometria...

- Geometria é assunto que aterroriza muito os maratonistas!
- Isso porque lidar com os todos os casos possíveis, e também os casos degenerados é geralmente trabalhoso.
- Se é necessário lidar com pontos flutuantes, nos confrotamos com mais problemas relacionados a precisão.
- Muitas vezes, os problemas de geometria na maratona geralmente são “simplificados”, e uma complexidade um pouco pior do que o esperado geralmente passa.
- De toda forma, qualquer enunciado com alguns pontos em um plano será imediatamente pulado por qualquer participante na primeira olhada.

Pontos

Representação de objetos geométricos

- É comum se criar objetos para representar pontos, segmentos, polígonos, para facilitar a manipulação.
- Ao invés de inventá-los, porém, vamos "reutilizar" uma abstração: Os números complexos.
- Assim, a gente não precisa inventar um monte de funções que precisamos para manipular pontos.
- Os números complexos já fazem parte da biblioteca padrão do C++, o que deixa tudo mais fácil.
- Os outros objetos podem ser representados por pares, ou vetores de pontos: Segmentos, polígonos, círculos, etc.

Números complexos!

- São da forma $z = a + bi$ onde $a, b \in \mathbb{R}$ e $i = \sqrt{-1}$.

Operações primitivas:

- Adição (`z1 + z2`): $z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$.
- Multiplicação escalar (`r * z`): $rz = ra + rbi$.
- Multiplicação complexa (`z1 * z2`):

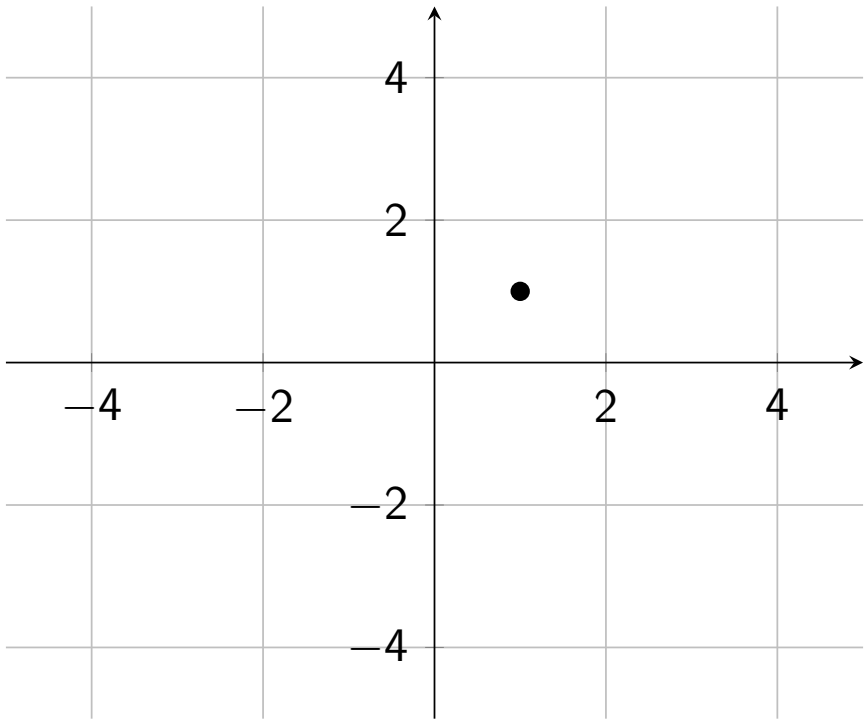
$$\begin{aligned} z_1 z_2 &= (a_1 + ib_1)(a_2 + ib_2) \\ &= a_1 a_2 + ia_1 b_2 + ib_1 a_2 + i^2 b_1 b_2 \\ &= a_1 a_2 + ia_1 b_2 + ib_1 a_2 - b_1 b_2 \\ &= (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + b_1 a_2) \end{aligned}$$

- Conjugado (`conj(z)`): $z^\star = a - bi$.
- Norma (`norm(z)`): $|z|^2 = a^2 + b^2 = zz^\star$.
- Valor absoluto (`abs(z)`): $|z| = \sqrt{a^2 + b^2}$.

Forma polar dos números complexos

- Algo interessante é ver a **forma polar** dos números complexos.
- Construção (`polar(r, th)`): $z = re^{\theta i}$.
- Obter θ (`arg(z)`) e r (`abs(z)`).

$$z = 1 + i = \sqrt{2}e^{\frac{\pi}{4}i}$$



Usos geométricos

- Produto escalar: `(conj(z1) * z2).px`.
- Produto vetorial: `(conj(z1) * z2).py`.
- Distância euclidiana ao quadrado: `norm(z1 - z2)`.
- Distância euclidiana: `abs(z1 - z2)`.
- Ângulo de elevação: `arg(b - a)`.
- Rotação na origem: `a * polar(1.0, th)`.
- Rotação no ponto pivô p: `(a-p) * polar(1.0, th) + p`.
- Ângulo ABC:
  `abs(remainder(arg(a-b) - arg(c-b), 2.0 * PI))`.

Lidando com ponto flutuante

Geralmente, vamos usar números complexos com `doubles`, então precisamos se atentar as peculiaridades. Para comparar com 0 por exemplo, declaramos um determinado valor pequeno, o ϵ que servirá como parâmetro para nossas comparações.

- $a = b$: `fabs(a - b) < EPS`.
- $x \geq 0$: `x > -EPS`.
- $x \leq 0$: `x < EPS`.

O GCC permite que os números complexos no C++ também possam ser implementados com números inteiros, geralmente com `long long`, mas é necessário ficar alerta porque muitas operações não vão funcionar direito (como por exemplo a distância euclidiana).

Biblioteca de pontos

Código `point.h`

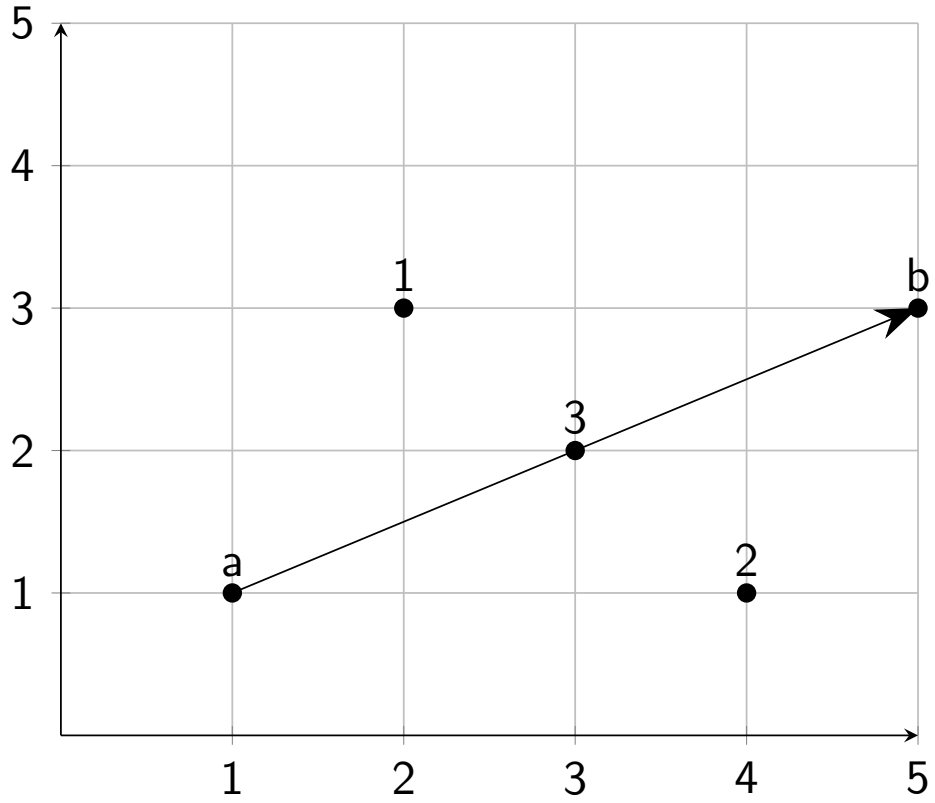
```
using pt = complex<double>;
#define px real()
#define py imag()
double dot(pt a, pt b) { return (conj(a) * b).px; }
double cross(pt a, pt b) { return (conj(a) * b).py; }
pt vec(pt a, pt b) { return b - a; }
int sgn(double v) { return (v > -EPS) - (v < EPS); }
// -1 (cw), 0 (colinear), +1 (ccw)
int seg_ornt(pt a, pt b, pt c) {
    return sgn(cross(vec(a, b), vec(a, c))); }
int ccw(pt a, pt b, pt c, bool col) {
    int o = seg_ornt(a, b, c);
    return (o == 1) || (o == 0 && col); }
const double PI = acos(-1);
double angle(pt a, pt b, pt c) {
    return abs(remainder(arg(a-b) - arg(c-b), 2.0 * PI)); }
```

Direção de três pontos

Código `triangle.cpp`

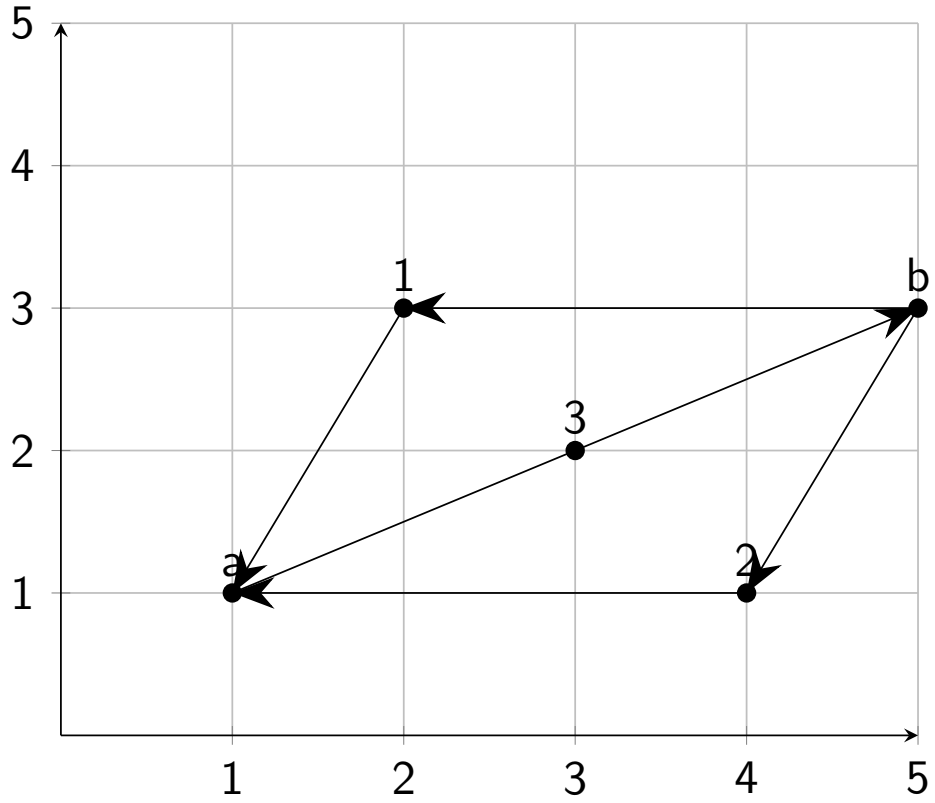
```
#include <bits/stdc++.h>
using namespace std;
const double EPS = 1e-9;
#include "point.h"
int main() {
    int t; cin >> t;
    while (t--) {
        int x1, y1, x2, y2, x3, y3;
        cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
        pt a (x1, y1), b (x2, y2), c (x3, y3);
        int ans = seg_ornt(a, b, c);
        if (ans == 0) { cout << "TOUCH\n"; }
        else if (ans == +1) { cout << "LEFT\n"; }
        else if (ans == -1) { cout << "RIGHT\n"; }
    }
}
```

Exemplo de direção de três pontos



Entrada	Saída
3 1 1 5 3 2 3 1 1 5 3 4 1 1 1 5 3 3 2	LEFT RIGHT TOUCH

Exemplo de direção de três pontos



Entrada	Saída
3 1 1 5 3 2 3 1 1 5 3 4 1 1 1 5 3 3 2	LEFT RIGHT TOUCH

Linhas e segmentos

Linhas e segmentos

Dois pontos fazem um segmento! Já as linhas são a extensão infinita de um segmento, então podemos representá-las por um segmento simbólico de comprimento irrelevante.
Agora alguns problemas com segmentos:

- Achar a interseção entre duas linhas que não são colineares
- Achar a interseção entre dois segmentos
- Verificar se um ponto está em um segmento
- Calcular a distância entre um ponto e um segmento
- Calcular a distância entre dois segmentos

Vamos ver tudo isso a seguir.

Biblioteca de segmentos

Código `seg.h`

```
#define aa first
#define bb second
using seg = pair<pt, pt>;
pt vec(seg s) { return vec(s.aa, s.bb); }
int ord_ornt(seg s, pt c) {
    return seg_ornt(s.aa, s.bb, c);
}
```

Verificar se ponto está no segmento

Código `onseg.h`

```
bool on_segment(seg s, pt p) {
    return cross(s.aa - p, s.bb - p) == 0
        && dot(s.aa - p, s.bb - p) <= 0;
}
```

Verificar se há intersecção entre dois segmentos

Código `intersectseg.h`

```
#include "onseg.h"
bool intersects_seg(seg s, seg t) {
    int o1 = ord_ornt(s, t.aa), o2 = ord_ornt(s, t.bb);
    int o3 = ord_ornt(t, s.aa), o4 = ord_ornt(t, s.bb);
    return (
        (o1 != o2 && o3 != o4) ||
        (o1 == 0 && on_segment(s, t.aa)) ||
        (o2 == 0 && on_segment(s, t.bb)) ||
        (o3 == 0 && on_segment(t, s.aa)) ||
        (o4 == 0 && on_segment(t, s.bb)));
}
```

Ponto mais próximo do segmento

Código `pointinseg.h`

```
pt point_in_seg(pt p, seg s) {
    if (s.aa == s.bb) { return s.aa; }
    double l = dot(vec(s.aa, p), vec(s)) / norm(vec(s));
    if (l < 0.0) { return s.aa; }
    if (l > 1.0) { return s.bb; }
    return s.aa + l*vec(s);
}
```

Distância entre ponto e segmento

Código `distsegpoint.h`

```
#include "pointinseg.h"
double distance_seg_point(pt p, seg s) {
    return abs(p - point_in_seg(p, s));
}
```

Distância entre segmento e segmento

Código `distsegseg.h`

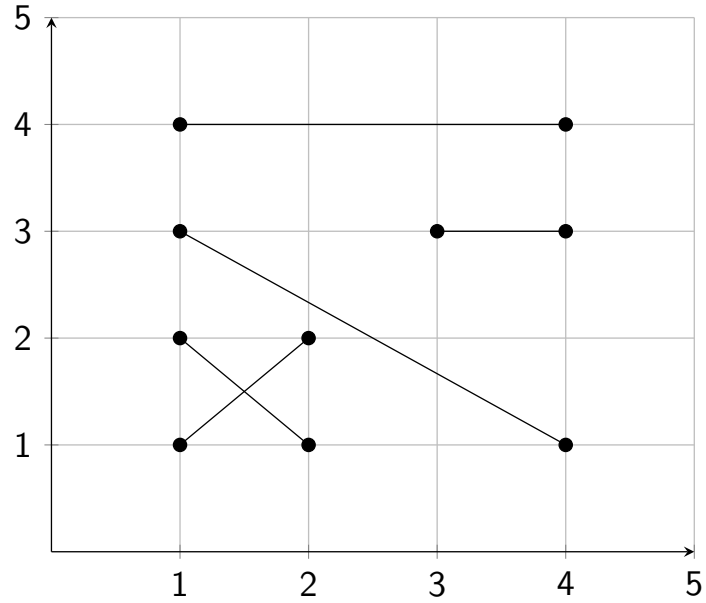
```
#include "intersectseg.h"
#include "distsegpoint.h"
double distance_seg_seg(seg s, seg t) {
    if (intersects_seg(s, t)) { return 0; }
    double v1 = distance_seg_point(s.aa, t);
    double v2 = distance_seg_point(s.bb, t);
    double v3 = distance_seg_point(t.aa, s);
    double v4 = distance_seg_point(t.bb, s);
    return min({ v1, v2, v3, v4 });
}
```

Utilização da distância entre segmento e segmento

Código `distsegseg.cpp`

```
#include <bits/stdc++.h>
using namespace std;
const double EPS = 1e-9;
#include "point.h"
#include "seg.h"
#include "distsegseg.h"
int main() {
    int t; cin >> t;
    while (t--) {
        int x1, y1, x2, y2, x3, y3, x4, y4;
        cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3 >> x4 >> y4;
        seg a = seg(pt(x1, y1), pt(x2, y2));
        seg b = seg(pt(x3, y3), pt(x4, y4));
        cout << distance_seg_seg(a, b) << "\n";
    }
}
```

Utilização da distância entre segmento e segmento (continuado)



Entrada	Saída
7 1 1 2 2 2 1 1 2 2 1 1 2 1 1 2 2 1 1 2 2 3 3 4 3 2 1 1 2 3 3 4 3 1 4 4 4 3 3 4 3 1 3 4 1 3 3 4 3 1 3 4 1 1 1 2 2	0 0 1.41421 2.12132 1 1.1094 0.27735

Polígonos

Polígonos

- Um conjunto ordenado de pontos.
- Podem estar em ordem horária ou anti-horária.
- Podem ser simples (sem buracos, e não intersecta consigo mesmo) ou complexos (permitem tudo isso)
- Geralmente convertemos polígonos complexos em polígonos simples para resolver problemas.

Problemas com polígonos

- Verificar se um ponto está dentro de um polígono.
- Partição de polígonos em outros polígonos:
  - Divisão em polígonos monotônicos.
  - Triangulação.
  - Problema da Galeria de Arte.
- Operações booleanas (união, intersecção, etc).
- Área de um polígono.
- Diagrama de Voronoi.

Área de um polígono

Algoritmo de Shoelace. Dá positivo ou negativo dependendo da ordem do polígono.

Código `polyarea.h`

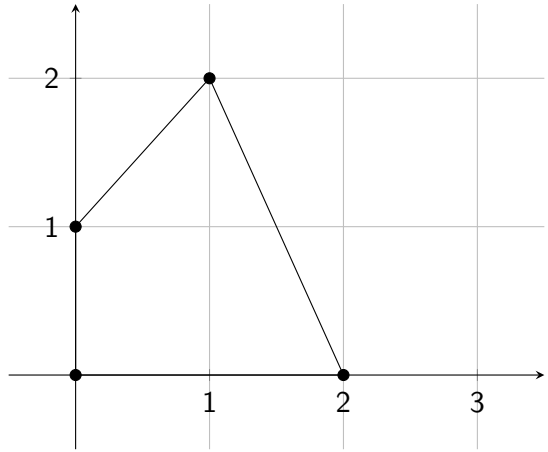
```
double polygon_area(vector<pt>& ps) {
    double area = cross(ps[ps.size()-1], ps[0]);
    for (int i = 1; i < ps.size(); i++) {
        area += cross(ps[i-1], ps[i]);
    }
    return area / 2.0;
}
```

Exemplo de área de um polígono

Código polyarea.cpp

```
#include <bits/stdc++.h>
using namespace std;
const double EPS = 1e-9;
#include "point.h"
#include "polyarea.h"
int main() {
    int n; cin >> n;
    vector<pt> ps (n);
    for (int i = 0; i < n; i++) {
        int x, y; cin >> x >> y; ps[i] = pt(x, y);
    }
    cout << polygon_area(ps) << "\n";
}
```

Exemplo de área de um polígono (continuado)



Entrada	Saída
4 2 0 1 2 0 1 0 0	2.5
4 0 0 0 1 1 2 2 0	-2.5

Teorema de Pick

- Um polígono simples com vértices em coordenadas inteiras.

$$A = i + \frac{b}{2} - 1$$

- A é a área de um polígono.
- i é o número de pontos de coordenada inteira no interior do polígono.
- b é o número de pontos de coordenada inteira na borda do polígono.

Polígonos convexos

- Um polígono convexo é aquele onde todo segmento entre dois vértices do polígono está contido na união do interior com a borda do polígono.
- Todos os ângulos internos são menores ou iguais a 180 graus.
- Polígonos estritamente convexos não permitem segmentos colineares.
- Se pegar todo conjunto de três pontos consutivos, eles sempre seguirão uma mesma direção ou serão colineares.

Verificar se ponto está dentro de um polígono convexo

Código `isinsidepolyconvex.h`

```
bool is_inside_poly(vector<pt>& ps, pt p) {
    int n = ps.size();
    double theta = 0;
    for (int i = 0; i < n; i++) {
        theta += (seg_ornt(ps[i], ps[(i+1)%n], p)
                >= 0 ? +1 : -1)
            * angle(ps[i], p, ps[(i+1)%n]);
    }
    return abs(theta - 2*PI) < EPS;
}
```

Verificar se ponto está dentro de um polígono

Código `isinsidepoly.h`

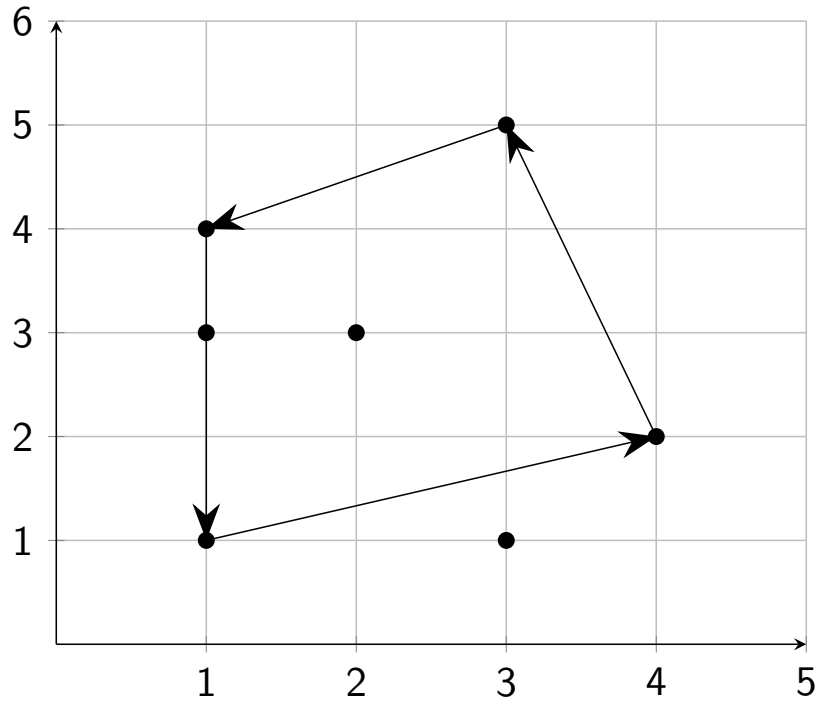
```
#include "onseg.h"
// 0 if outside, 1 if inside, strict if on boundary
int is_inside_poly(vector<pt>& ps, pt p, int strict) {
    int n = ps.size();
    if (n < 3) { return false; }
    int count = 0;
    for (int i = 0; i < n; i++) {
        int j = (i+1)%n;
        if (on_segment(seg(ps[i], ps[j]), p))
            return strict;
        count ^= (((p.py < ps[i].py) - (p.py < ps[j].py))
            * ord_ornt(seg(ps[i], ps[j]), p)) > 0;
    }
    return count;
}
```

Verificando pontos dentro do polígono

Código isinsidepoly.cpp

```
#include <bits/stdc++.h>
using namespace std; const double EPS = 1e-9;
#include "point.h"
#include "seg.h"
#include "isinsidepoly.h"
int main() {
    int n, m; cin >> n >> m; vector<pt> ps (n);
    for (int i = 0; i < n; i++) {
        int x, y; cin >> x >> y; ps[i] = pt(x, y); }
    while (m--) {
        int x, y; cin >> x >> y;
        switch (is_inside_poly(ps, pt(x, y), 2)) {
            case 0: cout << "OUTSIDE\n"; break;
            case 1: cout << "INSIDE\n"; break;
            case 2: cout << "BOUNDARY\n"; break;
        }
    }
}
```

Verificando pontos dentro do polígono (continuado)



Entrada	Saída
4 3 1 1 4 2 3 5 1 4 2 3 3 1 1 3	INSIDE OUTSIDE BOUNDARY

Técnicas Algorítimicas

Varredura linear

- Uma das técnicas principais em geometria computacional.
- A ideia é imaginar uma linha (frequentemente vertical) que varre o plano, parando em alguns pontos.
- As operações geométricas são restritas a objetos geométricos que intersectam ou estão próximos a linha de varredura quando ela para.
- Ao final da varredura, o problema está solucionado.

Fecho convexo

- Vamos fazer uma varredura linear para resolver o problema do fecho convexo.
- Algoritmo das cadeias monotônicas de Andrew.

Fecho convexo usando cadeias monotônicas

Código monotone.h

```
// O(n lg n)
vector<pt> convex_hull(vector<pt>& ps, bool col = false) {
    int k = 0, n = ps.size(); vector<pt> ans (2*n);
    sort(ps.begin(), ps.end(), [](pt a, pt b) {
        return make_pair(a.px, a.py) < make_pair(b.px, b.py); });
    for (int i = 0; i < n; i++) {
        while (k >= 2 && !ccw( /* lower hull */
            ans[k-2], ans[k-1], ps[i], col)) { k--; }
        ans[k++] = ps[i];
    }
    if (k == n) { ans.resize(n); return ans; }
    for (int i = n-2, t = k+1; i >= 0; i--) {
        while (k >= t && !ccw( /* upper hull */
            ans[k-2], ans[k-1], ps[i], col)) { k--; }
        ans[k++] = ps[i];
    }
    ans.resize(k-1); return ans;
}
```

Varredura angular

- A linha ao invés de varrer o plano em uma direção, varre o plano usando uma linha que vai rotacionando.
- Uma subtécnica são os *rotating calipers* que se ancoram no conceito de se rotacionar de forma a encontrar pontos que são opostos.

Fecho convexo com varredura angular

- Varremos o plano de acordo com o ângulo de um ponto extremo específico.
- Varredura de Graham.

Fecho convexo de Graham

Código graham.h

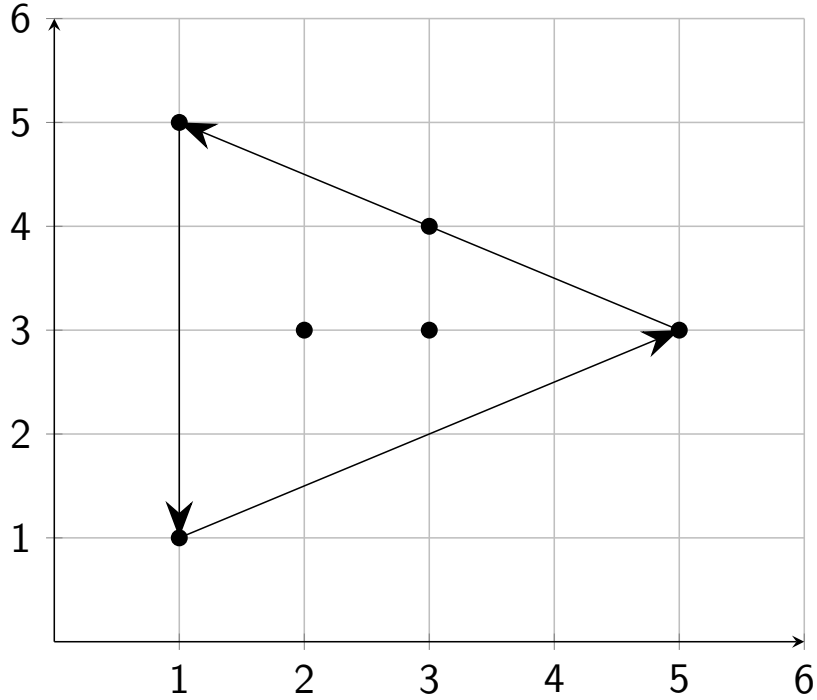
```
// O(n lg n)
vector<pt> convex_hull(vector<pt>& ps, bool col = false) {
    pt p0 = *min_element(ps.begin(), ps.end(), [](pt a, pt b) {
        return make_pair(a.py, a.px) < make_pair(b.py, b.px); });
    sort(ps.begin(), ps.end(), [&p0](pt a, pt b) {
        int o = seg_ornt(p0, a, b);
        return o < 0 || (o == 0 && norm(p0 - a) < norm(p0 - b)); });
    if (col) {
        int i = ps.size(); i--;
        while (i >= 0 && seg_ornt(p0, ps[i], ps.back()) == 0)
            i--;
        reverse(ps.begin()+i+1, ps.end());
    }
    vector<pt> ans; for (int i = 0; i < ps.size(); i++) {
        while (ans.size() > 1 && !ccw(
                ps[i], ans.back(), ans[ans.size()-2], col))
            ans.pop_back();
        ans.push_back(ps[i]);
    } return ans;
}
```

Utilização do fecho convexo

Código convex.cpp

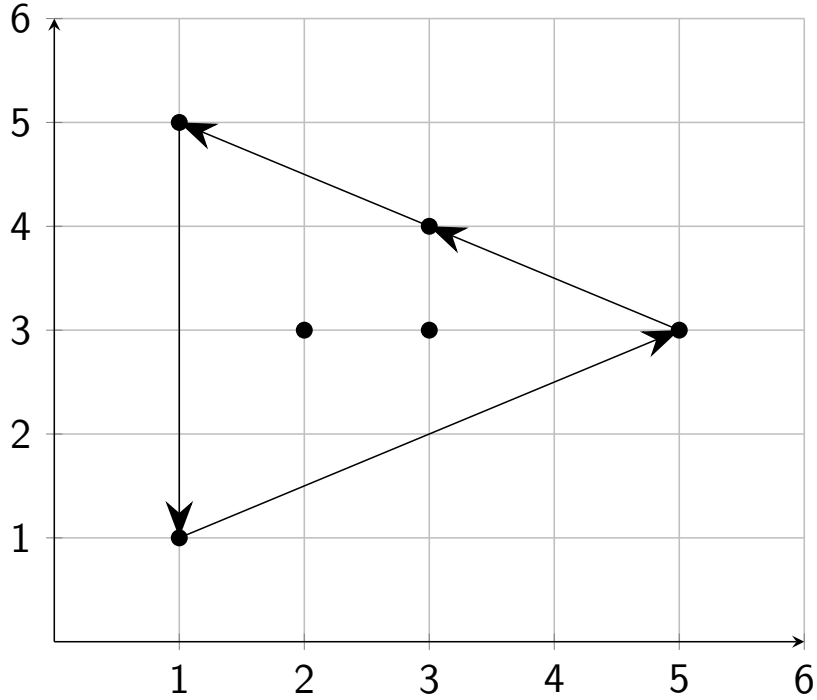
```
#include <bits/stdc++.h>
using namespace std; const double EPS = 1e-9;
#include "point.h"
#include "monotone.h"
int main() {
    int n, c; cin >> n >> c; vector<pt> ps (n);
    for (int i = 0; i < n; i++) {
        int x, y; cin >> x >> y; ps[i] = pt(x, y);
    }
    vector<pt> hull = convex_hull(ps, c);
    cout << hull.size() << "\n";
    for (pt p : hull)
        cout << int(p.px) << " " << int(p.py) << "\n";
}
```

Utilização do fecho convexo (continuado)



Entrada	Saída
6 0 1 1 1 5 2 3 3 3 3 4 5 3	3 1 1 5 3 1 5

Utilização do fecho convexo (continuado)



Entrada	Saída
6 1 1 1 1 5 2 3 3 3 3 4 5 3	4 1 1 5 3 3 4 1 5

E isso é tudo!

- Todo o conteúdo que vocês vão precisar está aqui!
- Tem uma matéria inteira sobre Geometria Computacional.
- Confira alguns links com leituras extras.
- Geometria com números complexos (`https://codeforces.com/blog/entry/22175`).
- Diagrama de Voronoi (`https://codeforces.com/blog/entry/85638`).
- Fecho convexo [escrito pelo Fernando] (`https://cp-algorithms.com/geometry/convex-hull.html`).
- Diagrama de Voronoi e Triangulação de Delaunay (`https://codeforces.com/blog/entry/85638`).
- Bons estudos!