

Introdução

Template

```
#include <bits/stdc++.h>
using namespace std;
#define DEBUG true
#define coutd if(DEBUG) cout
#define debugf if(DEBUG) printf
#define ff first
#define ss second
#define len(x) int(x.size())
#define all(x) x.begin(), x.end()
#define pb push_back
using ll = long long;
using ii = pair<int, int>;
using vi = vector<int>;
const double PI = acos(-1);
const double EPS = 1e-9;

int main(){
    ios_base::sync_with_stdio(0);

}
```

Matemática

Números primos

Descrição

- sieve: Crivo de Eristótenes, armazena no vetor primes os primos até **n** e no bitset bs se o número é ou não primo. em complexidade **$O(n \log \log n)$** ;
- is_prime_sieve: Calcula um número primo caso **sievesize** seja maior ou igual que \sqrt{N} em complexidade **$O(\sqrt{n}/\log n)$**
- is_prime: Calcula se um número é ou não primo de maneira rápida sem depender do crivo em complexidade **$O(\sqrt{n})$**

```
using ll = long long;
const long long MAX = 100000009;
ll sievesize;
bitset<MAX> bs;
vector<ll> primes;

void sieve(ll n){
    sievesize = n+1;
    bs.set();
    bs[0]=bs[1]=0;
    for(ll i=2; i<=sievesize; ++i){
        if(bs[i]){
            for(ll j=i*i; j<=sievesize; j+=i)
                bs[j]=0;
            primes.push_back(i);
        }
    }
}

bool is_prime_sieve(ll n){
    if(n<=(ll)sievesize) return bs[n];
    for(size_t i=0; i<primes.size() and primes[i]*primes[i]<=n; ++i)
        if(n%primes[i] == 0) return false;
    return true;
}

bool is_prime(ll n){
    if(n<0) n=-n;
    if(n<5 or n%2==0 or n%3==0)
        return (n==2 or n==3);
    ll maxP = sqrt(n)+2;
    for(ll p=5; p<maxP; p+=6){
        if( p<n and n%p==0 ) return false;
        if( p+2<n and n%(p+2)==0 ) return false;
    }
    return true;
}
```

Aritmética modular

Descrição

- gcd: Calcula maior divisor comum em complexidade $O(\log a + \log b)$
- lcm: Calcula o menor múltiplo comum em complexidade $O(\log a + \log b)$
- ext_gcd: Algoritmo estendido de euclides complexidade: $O(\log a + \log b)$
- num_div: Calcula o números de divisores de N em complexidade $O(\sqrt{n})$
- mod_inv: Calcula inverso modular de a em módulo m complexidade: $O(\log a + \log b)$
- mod_exp: Calcula a elevado à b em módulo m em complexidade: $O(\log b)$
- mod_mul: Calcula $(a*b)\%m$ sem overflow em complexidade: $O(\log a + \log b)$
- diophantine: Acha uma solução na equação no formato $ax+by=c$, sendo x e y as incognitas; Após a divisão de a, b e c pelo mdc(a, b) as outras soluções são $x=x_0+bt$, $y=y_0-at$; complexidade: $O(\log a + \log b)$
- preprocess_fat: Pré-processa os fatoriais em módulo m até MAXN em complexidade $O(MAXN)$

```
using ll = long long;
template<typename T>
T gcd(T a, T b){
    return b==0 ? a : gcd(b, a%b);
}
template<typename T>
T lcm(T a, T b){
    return a*(b/gcd(a, b));
}
template<typename T>
T ext_gcd(T a, T b, T& x, T& y){
    if(b==0){
        x=1; y=0; return a;
    }
    else{
        T g = ext_gcd(b, a%b, y, x);
        y-=a/b*x; return g;
    }
}
template<typename T>
T num_div(T n){
    T ans=0;
    for(T i=1; i*i<=n; ++i)
        if(n%i == 0)
            ans+=( i==n/i ? 1 : 2);
    return ans;
}
template<typename T>
T mod_inv(T a, T m){
    T x, y;
    ext_gcd(a, m, x, y);
    return (x%m+m)%m;
}
```

```

template<typename T>
T mod_mul(T a, T b, T m){
    T x=0, y=a%m;
    while(b>0){
        if(b%2==1) x=(x+y)%m;
        y=(y*2)%m; b/=2;
    }
    return x%m;
}

template<typename T>
T mod_exp(T a, T b, T m){
    if(b == 0) return (T) 1;
    T c = mod_exp(a, b/2, m);
    c = (c*c)%m;
    if(b%2 != 0) c=(c*a)%m;
    return c;
}

template<typename T>
void diophantine(T a, T b, T c, T& x, T& y){
    T d = ext_gcd(a, b, x, y);
    x *= c/d; y*= c/d;
}

#define MAXN 1000009
using ll = long long;
ll fat[MAXN];
void preprocess_fat(ll m){
    fat[0] = 1;
    for(ll i=1; i<MAXN; ++i)
        fat[i]=(i*fat[i-1])%m;
}

```

Grafos

Travessias

Descrição

- dfs: (Depth-First Search) Travessia por profundidade complexidade **$O(N)$** sendo N o número de arestas do vértice u
- bfs: (Breadth-First Search) Travessia por largura; Armazena no vetor dist, a distância do vertice u para os demais vertice; o próprio vértice e vértices que não foram visitados tem distância 0; complexidade **$O(N)$** sendo N o número de arestas do vértice u


```
#define MAXN 1009
bitset<MAXN> visited;
vector<int> adj_list[MAXN];

void dfs(int u){
    visited[u]=true;
    for(auto v:adj_list[u]){
        if(not visited[v]){
            dfs(v);
        }
    }
}

int dist[MAXN];
void bfs(int u){
    queue<int> q;
    q.push(u);
    visited[u]=true;
    memset(&dist, 0, sizeof dist);
    while(not q.empty()){
        auto x = q.front();
        q.pop();
        for(auto v:adj_list[x]){
            if(visited[v]) continue;

            visited[v]=true;
            dist[v] = dist[x]+1;
            q.push(v);
        }
    }
}
```

Caminhos mínimos

Calcula o caminho mínimo de **s** para **t** em um grafo com **n** arestas numeradas de **1 a n**. Armazena em  **a** aresta antecessora a **x**, desse modo pode-se recuperar o caminho mínimo.

```
//Macros
#define ff first
#define ss second
using ii = pair<int, int>;
const int INF = 1e9;
```

Algoritmo de Bellman-Ford

Tambem pode verificar se o grafo tem ciclo negativo; Complexidade **O(VE)**

```
#define MAXN 1009
vector<ii> adj_list[MAXN];
int dist[MAXN];
int pred[MAXN];

int bellmanford(int s, int t, int n){
    for(int i=1; i<=n; ++i){
        dist[i]=INF;
        pred[i]=-1;
    }
    dist[s]=0; pred[s]=s;
    bool has_negative_cicle = false;
    for(int i=1; i<=n; ++i){
        for(int u=1; u<=n; ++u){
            for(auto x:adj_list[u]){
                auto v=x.ff, w=x.ss;
                if(i==n and dist[v]>dist[u]+w)
                    has_negative_cicle=true;
                else{
                    if(dist[v]>dist[u]+w){
                        dist[v] = dist[u]+w;
                        pred[v] = u;
                    }
                }
            }
        }
    }
    return dist[t];
}
```

Shortest Path Faster Algorithm (SPFA)

Otimização do algoritmo de Bellman-Ford. Retorna **-1** se o grafo tiver ciclo negativo..Complexidade média igual ao algoritmo de dijkstra, no pior caso **O(VE)**;

```
#define MAXN 100009
vector<ii> adj_list[MAXN];
int dist[MAXN];
bool inq[MAXN];
int vis[MAXN];
int pred[MAXN];

int spfa(int s, int t, int n){
    for(int i=1; i<=n; ++i){
        dist[i] = INF;
        pred[i]=-1;
    }
    memset(&inq, false, sizeof inq);
    memset(&vis, 0, sizeof vis);
    dist[s]=0; pred[s]=s;
    queue<int> q;
    q.push(s);
    inq[s]=true;
    while(not q.empty()){
        int u=q.front(); q.pop();
        if(vis[u]>n) return -1;
        inq[u] = false;
        for(auto x:adj_list[u]){
            auto v=x.ff, w=x.ss;
            if(dist[u]+w<dist[v]){
                dist[v]=dist[u]+w;
                pred[v]=u;
                if(not inq[v]){
                    ++vis[v]; q.push(v);
                    inq[v] = true;
                }
            }
        }
    }
    return dist[t];
}
```

Algoritmo de Dijkstra

Tem complexidade melhor que os outros algoritmos porém, **assume que o grafo não contém ciclo negativo**, essa implementação aceita arestas negativas; Complexidade $O(V \log V + E)$

```
#define MAXN 100009
vector<ii> adj_list[MAXN];
int dist[MAXN];
int pred[MAXN];
bitset<MAXN> visited;

int dijkstra(int s, int t, int n){
    for(int i=1; i<=n; ++i){
        dist[i]=INF;
        pred[i]=-1;
    }
    visited.reset();
    dist[s]=0; pred[s]=s;
    set<ii> pq;
    pq.insert(ii(0, s));
    while(not pq.empty()){
        auto u = pq.begin()->second;
        pq.erase(pq.begin());
        if(visited[u]) continue;
        for(auto x:adj_list[u]){
            auto v=x.ff, w=x.ss;
            if(dist[v]>dist[u]+w){
                pq.erase(ii(dist[v], v));
                dist[v]=dist[u]+w;
                pred[v]=u;
                pq.insert(ii(dist[v], v));
            }
        }
    }
    return dist[t];
}
```


Estrutura de dados

Fenwick Tree

Todas as implementações abaixo são indexadas em 1 portanto **não deve ser feito a consulta do índice 1**

Range Sum Query em $O(\log n)$ e atualização com soma pontual

```
class BITree {
private:
    vector<long long> ts;
    size_t N;

    int LSB(int n) { return n&(-n); }

    long long RSQ(int i){
        long long sum = 0;

        while(i>=1){
            sum+=ts[i];
            i-=LSB(i);
        }
        return sum;
    }
public:
    BITree(size_t n) : ts(n+1, 0), N(n) {};

    long long RSQ(int i, int j){
        return RSQ(j) - RSQ(i-1);
    }

    void add(size_t i, const long long& x){
        if(i==0) return;
        while(i<=N){
            ts[i]+=x;
            i+=LSB(i);
        }
    }
};
```

Range update em $O(\log n)$ query pontual

```
class BITree {
private:
    vector<long long> ts;
    size_t N;

    int LSB(int n) { return n&(-n); }

    void add(size_t i, ll x){
        while (i <= N) {
            ts[i] += x;
            i += LSB(i);
        }
    }
    long long RSQ(int i){
        long long sum = 0;

        while(i>=1){
            sum+=ts[i];
            i-=LSB(i);
        }
        return sum;
    }
public:
    BITree(size_t n) : ts(n+1, 0), N(n) {};

    ll value_at(int i) { return RSQ(i); }

    void range_add(size_t i, size_t j, long long x){
        add(i, x);
        add(j+1, -x);
    }
};
```

Query bidimensional em $O(\log n.m)$ update com soma pontual

```
class BITree2D{
private:
    size_t rows;
    size_t columns;
    vector<vector<long long>> ft;

    int LSB(long long n) { return n&(-n); }

    long long RSQ(int y, int x){
        long long sum = 0;

        for(int i=y; i>0; i-=LSB(i))
            for(int j=x; j>0; j-=LSB(j))
                sum+=ft[i][j];

        return sum;
    }
public:
    BITree2D(size_t y, size_t x) : ft(y+1, vector<long long>(x+1, 0)),
rows(y), columns(x) {}

    // Y e X sao as coordenadas do ponto superior
    // x e y sao as coordenadas do ponto inferior
    // RSQ vai retornar a soma do quadrado formado pelos pontos
    long long RSQ(int y, int x, int Y, int X){
        return RSQ(Y, X) - RSQ(Y, x-1) - RSQ(y-1, X) + RSQ(y-1, x-1);
    }

    void add(int y, int x, long long v){
        for(int i=y; i<=rows; i+=LSB(i))
            for(int j=x; j<=columns; j+=LSB(j))
                ft[i][j] += v;
    }
};
```

Range Product Query em $O(\log n)$ e update com multiplicação pontual

```
class BITree{
private:
    size_t N;
    vector<long long> ft;
    vector<int> sz; // 1 se o indice i for 0

    int LSB(const long long& n) { return n&(-n); }

    long long RPQ(int i){
        long long prod = 1;
        while(i){
            prod*=ft[i];
            i-=LSB(i);
        }
        return prod;
    }

    // Funções rsq para o vetor com zeros
    int RSQ(int i){
        int sum = 0;
        while(i){
            sum+=sz[i];
            i-=LSB(i);
        }
        return sum;
    }

    int RSQ(int i, int j){
        return RSQ(j)-RSQ(i-1);
    }

    void multiply(int i, long long k){
        while(i <= N){
            ft[i] *= k;
            i+=LSB(i);
        }
    }

    void add(int i, int k){
        while(i <= N){
            sz[i] += k;
            i+=LSB(i);
        }
    }

public:

    BITree(int n) : N(n), ft(n+1, 1), sz(n+1, 0) { }
```

```
long long RPQ(int i, int j){
    long long p = RPQ(j)/RPQ(i-1);
    int z = RSQ(i, j);

    if(z) return 0;
    else return p;
}

void update(int i, const long long& k){
    if(k)
        multiply(i, k);
    else if (RSQ(i, i)==0)
        add(i, 1);
}

};
```

Paradigmas de programação

Pares de pontos mais próximos

```
const double EPS = 1e-9;
const double INF = 1e18;
struct point{
    double x;
    double y;

    point() { x=y=0.0; }
    point(double _x, double _y): x(_x), y(_y) {}

    bool operator <(point other) const{
        return fabs(x-other.x)<EPS ? y<other.y : x<other.x;
    }
};
double dis(point a, point b){
    return hypot(a.x-b.x, a.y-b.y);
}

pair<point, point> closet_pair(vector<point>& vs){
    sort(vs.begin(), vs.end());
    int n = vs.size();
    set<point> st;
    double d = INF;
    auto closest = make_pair(point(), point());
    st.insert(point(vs[0].y, vs[0].x));
    for(int i=1; i<n; ++i){
        auto p = vs[i];
        auto it = st.lower_bound(point(p.y-d, -INF));
        while(it != st.end()){
            auto q = point(it->y, it->x);
            if(q.x < p.x-d){
                it = st.erase(it);
                continue;
            }
            else if(q.y > p.y+d) break;
            auto t = dis(p, q);
            if(t<d){
                d = t;
                closest = make_pair(p, q);
            }
            ++it;
        }
        st.insert(point(p.y, p.x));
    }
    return closest;
}
```

Programação dinâmica

Knapsack

Problema clássico da mochila com pesos e valores dos itens, no código abaixo w é o peso que a pessoa pode carregar, wt é um array de pesos de tamanho n , val é um array de valores de tamanho n , e n é o tamanho dos arrays.

- A matriz do `knapSack` foi inicializada globalmente para poupar tempo de processamento inicializando os valores zerados
- Com esse código é necessário declarar os valores máximos que a matriz pode assumir como constantes

```
const int MAXN = 20000; // Número máximo de itens
const int MAXW = 20000; // Número máximo de peso carregado
int mt[MAXN+1][MAXW+1]; // Matriz global é iniciada com zeros

int knapSack(int w, int wt[], int val[], int n){
    for(int i=1; i<=n; i++){
        for(int j=1; j<=w; j++){
            if(wt[i-1]<=j)
                mt[i][j] = max(val[i-1]+mt[i-1][j-wt[i-1]], mt[i-1][j]);
            else
                mt[i][j] = mt[i-1][j];
        }
    }
    return mt[n][w];
}
```

Geometria Computacional

Ponto 2D / Vetor

Representação de pontos e vetores como pontos.

Estrutura

Descrição

- **(double)** norm: Calcula norma do vetor em relação a origem
- **(double)** normalized: Retorna vetor normalizado (Norma 1)
- **(double)** angle: Angulo do vetor com a origem
- **(double)** polarAngle: Angulo polar em relação a origem

```
const double EPS = 1e-9;
const double PI = acos(-1);

struct point{
    double x, y;

    point(double _x, double _y) : x(_x), y(_y) {} ;
    point(){ x=y=0.0; };

    bool operator <( point other ) const {
        if(fabs( x-other.x )>EPS) return x<other.x;
        else return y < other.y;
    }
    bool operator ==(point other) const {
        return fabs(x-other.x) < EPS && fabs(y-other.y)<EPS;
    }
    point operator +(point other) const {
        return point( x+other.x, y+other.y);
    }
    point operator -(point other) const {
        return point( x-other.x, y-other.y );
    }
    point operator *(double k) const {
        return point(x*k, y*k);
    }

    double norm() { return hypot(x, y); }
    point normalized(){ return point(x, y)*(1.0/norm()); }
    double angle() { return atan2(y, x); }
    double polar_angle(){
        double a = atan2(y, x);
        return a < 0 ? a + 2*PI : a;
    }
};
```


Funções

Descrição

- **(double)** dist: Distância entre 2 pontos
- **(double/int)** inner: Produto interno entre 2 vetores
- **(double/int)** cross: Produto vetorial entre 2 vetores (componente z)
- **(double/int)** ccw: Verifica se os pontos estão no sentido antihorário (DISCRIMINANTE)
- **(double/int)** collinear: Teste de colinearidade entre os pontos
- **(double)** rotate: Rotação do ponto em relação a origem
- **(double)** angle: Angulo formado entre vetores a e b com o de origem
- **(double)** proj: Projeção de u sobre v
- **(double/int)** between: Verifica se o ponto q está dentro no segmento p r
- **(double)** line_intersect: Retorna ponto formado pela intersecção das retas p q e A B **Se as retas forem colineares c é zero e a função retorna nan**
- **(double/int)** parallel: Teste de paralelidade
- **(double)** seg_intersects: Verifica se segmentos a b e p q se interceptam
- **(double)** closet_point: Ponto mais próximo entre p e o segmento de reta a b

```
double dist(point p1, point p2){
    return hypot(p1.x-p2.x, p1.y-p2.y);
}
double inner(point p1, point p2){
    return p1.x*p2.x+p1.y*p2.y;
}
double cross(point p1, point p2){
    return p1.x*p2.y-p1.y*p2.x;
}
bool ccw(point p, point q, point r){
    return cross(q-p, r-p)>0;
}
bool collinear(point p, point q, point r){
    return fabs(cross(q-p, r-p))<EPS;
}
point rotate(point p, double rad){
    return point(p.x*cos(rad)-p.y*sin(rad),
                p.x*sin(rad)+p.y*cos(rad));
}
double angle(point a, point o, point b){
    return acos(inner(a-o, b-o)/(dist(o, a)*dist(o,b)));
}
point proj(point u, point v){
    return v*(inner(u, v)/inner(v, v));
}
bool between(point p, point q, point r){
    return collinear(p, q, r) && inner(p-q, r-q)<=0;
}
```

```

point line_intersect(point p, point q, point A, point B){
    double c = cross(A-B, p-q);
    double a = cross(A, B);
    double b = cross(p, q);
    return ( (p-q)*(a/c) ) - ( (A-B)*(b/c) );
}
bool parallel(point a, point b){
    return fabs(cross(a, b))<EPS;
}
bool seg_intersects(point a, point b, point p, point q){
    if(parallel(a-b, p-q)){
        return between(a, p, q) || between(a, q, b) ||
               between(p, a, q) || between(p, b, q);
    }
    point i = line_intersect(a, b, p, q);
    return between(a, i, b) && between(p, i, q);
}
point closet_point(point p, point a, point b){
    double u = inner(p-a, b-a)/inner(b-a, b-a);
    if(u < 0.0) return a;
    if(u > 1.0) return b;
    return a+((b-a)*u);
}

```

Polígono 2D

O polígono é representado por um vetor de pontos portanto não tem estrutura.

Funções

Descrição

- **(double/int)** signed_area: Retorna área do polígono (Pode dar valor negativo)
- **(double/int)** area: Retorna área positiva do polígono
- **(double/int)** left_index: Índice do ponto mais a esquerda
- **(double/int)** make_polygon: Coloca o índice do ponto mais a esquerda como 0 (Necessário para as funções que chamam polygon)
- **(double)** perimeter: Calcula o perímetro do polígono
- **(double/int)** is_convex: Verifica se polígono é convexo
- **(double)** in_polygon: Verifica se um ponto está no polígono
- **(double)** cut_polygon: Polígono a direita que é formado pelo corte do polígono P pela reta formada pelos pontos a e b

```
using polygon = vector<point>;
const double PI = acos(-1);

double signed_area(polygon &P){
    double result = 0.0;
    int n = P.size();
    for(int i=0; i<n; i++){
        result+=cross(P[i], P[(i+1)%n]);
    }
    return result/2.0;
}

double area(polygon &P){
    return fabs(signed_area(P));
}

int left_index(vector<point>& P){
    int ans = 0;
    for(int i=1; i<int(P.size()); i++){
        if (P[i]<P[ans]) ans = i;
    }
    return ans;
}

polygon make_polygon(vector<point> P){
    if(signed_area(P)<0.0)
        reverse(P.begin(), P.end());

    int li = left_index(P);
    rotate(P.begin(), P.begin()+li, P.end());
    return P;
}
```

```

double perimeter(polygon &P){
    double result = 0.0;
    int n = P.size();
    for(int i=0; i<n; i++) result+=dist(P[i], P[(i+1)%n]);
    return result;
}

bool is_convex(polygon& P){
    int n = P.size();
    if( n<3 ) return false;
    bool left = ccw(P[0], P[1], P[2]);
    for(int i=1; i<n; i++){
        if(ccw(P[i], P[(i+1)%n], P[(i+2)%n]) != left)
            return false;
    }
    return true;
}

bool in_polygon(polygon &P, point p){
    if (P.size() == 0u) return false;
    double sum = 0.0;
    int n = P.size();
    for(int i=0; i<n; i++){
        if(P[i] == p || between(P[i], p, P[(i+1)%n]))
            return true;
        if(ccw(p, P[i], P[(i+1)%n]))
            sum+=angle(P[i], p, P[(i+1)%n]);
        else
            sum-=angle(P[i], p, P[(i+1)%n]);
    }
    return fabs(fabs(sum)-2*PI) < EPS;
}

polygon cut_polygon(polygon &P, point a, point b){
    vector<point> R;
    double left1, left2;
    int n = P.size();
    for(int i=0; i<n; i++){
        left1 = cross(b-a, P[i]-a);
        left2 = cross(b-a, P[(i+1)%n]-a);
        if (left1 > -EPS) R.push_back(P[i]);
        if (left1 * left2 < -EPS)
            R.push_back(line_intersect(P[i], P[(i+1)%n], a, b));
    }
    return make_polygon(R);
}

```

Convex Hull

Dado um conjunto de pontos retorna o polígono que contém todos os pontos em $O(n \log n)$. Caso precise considerar os pontos no meio de uma aresta trocar ccw para ≥ 0 . CUIDADO: Se todos os pontos forem colineares, vai dar RTE.

```
point pivot(0, 0);

bool angle_cmp(point a, point b){
    if(collinear(pivot, a, b))
        return inner(pivot-a, pivot-a) < inner(pivot-b, pivot-b);
    return cross(a-pivot, b-pivot) >= 0;
}

polygon convex_hull(vector<point> P){
    int i, j, n = P.size();
    if( n<=2 ) return P;
    int P0 = left_index(P);
    swap(P[0], P[P0]);
    pivot = P[0];
    sort(++P.begin(), P.end(), angle_cmp);
    vector<point> S;
    S.push_back(P[n-1]);
    S.push_back(P[0]);
    S.push_back(P[1]);
    for(i=2; i<n;){
        j=int(S.size()-1);
        if(ccw(S[j-1], S[j], P[i]))
            S.push_back(P[i++]);
        else S.pop_back();
    }
    reverse(S.begin(), S.end());
    S.pop_back();
    reverse(S.begin(), S.end());
    return S;
}
```

Monotone chain

- Com esse algoritmo é possível conseguir o polígono que contém todos os pontos em $O(n \log n)$ com todos os pontos consistindo como inteiros e não ponto flutuante. **Caso precise considerar os pontos no meio de uma aresta trocar ccw para ≥ 0**
- Nessa rotina é preciso usar a struct point como **long long**.

```
using polygon = vector<point>;

polygon convex_hull(const vector<point> points){
    vector<point> P(points);
    sort(P.begin(), P.end());
    vector<point> lower, upper;
    for(const auto& p: P){
        int n = int(lower.size());
        while(n>=2 and ccw(lower[n-2], lower[n-1], p)){
            lower.pop_back();
            n = int(lower.size());
        }
        lower.push_back(p);
    }
    reverse(P.begin(), P.end());
    for(const auto& p: P){
        int n = int(upper.size());
        while(n>=2 and ccw(upper[n-2], upper[n-1], p)){
            upper.pop_back();
            n = int(upper.size());
        }
        upper.push_back(p);
    }
    lower.pop_back();
    lower.insert(lower.end(), upper.begin(), upper.end()-1);
    return lower;
}
```

Círculo 2D

Estrutura

Descrição

- **(double)** area: Calcula área do círculo
- **(double)** chord: Calcula o comprimento da corda com ângulo em radianos
- **(double)** sector: Calcula o comprimento do setor com ângulo radianos
- **(double)** intersects: Verifica se os círculos se interceptam
- **(double)** contains: Verifica se um ponto p está dentro do círculo
- **(double)** tangent_points: Retorna os 2 pontos que formam as retas com p tangentes ao círculo, (A função asin retorna nan se o ponto estiver dentro do círculo)
- **(double)** intersection_points: Retorna os 2 pontos de intersecção entre os círculos

```
const double PI = acos(-1);
const double EPS = 1e-9;
struct circle{
    point c;
    double r;

    circle() { c=point(); r=0; }
    circle(point _c, double _r) : c(_c), r(_r) {}

    double area() { return PI*r*r; }
    double chord (double rad){ return 2*r*sin(rad/2.0); }
    double sector (double rad) { return 0.5*rad*area()/PI; }
    bool intersects(circle other){
        return dist(c, other.c) < r+other.r;
    }
    bool contains(point p) { return dist(c, p) <= r + EPS; }
    pair<point, point> tangent_points( point p ){
        double d1 = dist(p, c), theta = asin(r/d1);
        point p1 = rotate(c-p, -theta);
        point p2 = rotate(c-p, theta);
        p1 = p1*(sqrt(d1*d1-r*r)/d1)+p;
        p2 = p2*(sqrt(d1*d1-r*r)/d1)+p;
        return make_pair(p1, p2);
    }
    pair<point, point> intersection_points(circle other){
        assert(intersects(other));
        double d = dist(c, other.c);
        double u = acos((other.r*other.r + d*d - r*r)/(2*other.r*d));
        point dc = ((other.c - c).normalized()) * r;
        return make_pair(c + rotate(dc, u), c + rotate(dc, -u));
    }
};
```

Funções

Descrição

- **(double)** inside_circle: Retorna 0 caso o ponto esteja dentro, 1 caso esteja na borda e 2 caso esteja fora do círculo
- **(double)** circumcircle: Círculo circunscrito no triângulo
- **(double)** incircle: Círculo inscrito do triângulo

```
int inside_circle(point p, circle c) {
    if (fabs(dist(p, c.c) - c.r)<EPS) return 1;
    else if(dist(p, c.c) < c.r) return 0;
    else return 2;
} // 0 = inside/ 1 = border/ 2 = outside
circle circumcircle(point a, point b, point c) {
    circle ans;
    point u = point((b-a).y, -(b-a).x);
    point v = point((c-a).y, -(c-a).x);
    point n = (c-b)*0.5;
    double t = cross(u, n)/cross(v, u);
    ans.c = ((a+c)*0.5) + (v*t);
    ans.r = dist(ans.c, a);
    return ans;
}
circle incircle( point p1, point p2, point p3){
    double m1 = dist(p2, p3);
    double m2 = dist(p1, p3);
    double m3 = dist(p1, p2);
    point c = (p1*m1+p2*m2+p3*m3)*(1/(m1+m2+m3));
    double s = 0.5*(m1+m2+m3);
    double r = sqrt(s*(s-m1)*(s-m2)*(s-m3))/s;
    return circle(c, r);
}
```


Triângulo 2D

Estrutura

Descrição

- **(double)** perimeter: Calcula perímetro do triângulo
- **(double)** semiPerimeter: Calcula semi-perímetro do triângulo
- **(double)** area: Calcula área do triângulo
- **(double)** r_incircle: Calcula o raio do círculo inscrito no triângulo
- **(double)** in_circle: Retorna o círculo inscrito do triângulo
- **(double)** r_circumcircle: Calcula raio do círculo circunscrito no triângulo
- **(double)** circum_circle: Retorna o círculo circunscrito do triângulo
- **(double/int)** is_inside: Retorna 0 caso o ponto esteja dentro do triângulo, 1 caso esteja na borda e 2 caso esteja fora

```
struct triangle{
    point a, b, c;
    triangle() { a=b=c=point(); }
    triangle(point _a, point _b, point _c) : a(_a), b(_b), c(_c) {}
    double perimeter() { return dist(a, b) + dist(b, c) + dist(c, a); }
    double semiPerimeter() { return perimeter()/2.0; }
    double area(){
        double s = semiPerimeter(), ab = dist(a, b), bc = dist(b, c), ca =
dist(c, a);
        return sqrt(s*(s-ab)*(s-bc)*(s-ca));
    }
    double r_incircle() {
        return area()/semiPerimeter();
    }
    circle in_circle() {
        return incircle(a, b, c);
    }
    double r_circumcircle() {
        return dist(a, b)*dist(b, c)*dist(c, a)/(4.0*area());
    }
    circle circum_circle(){
        return circumcircle(a, b, c);
    }
    int is_inside(point p){
        double u = cross(b-a, p-a)*cross(b-a, c-a);
        double v = cross(c-b, p-b)*cross(c-b, a-b);
        double w = cross(a-c, p-c)*cross(a-c, b-c);
        if (u>0.0 and v>0.0 and w>0.0) return 0;
        if (u<0.0 or v<0.0 or w<0.0) return 2;
        else return 1;
    }
} // 0 = inside/ 1 = border/ 2=outside
};
```