Biblioteca de Gabriel Pessoa :)

Contents

1 String Algorithms

1.1 String Alignment

```
int pd[ms][ms];

int edit_distance(string &a, string &b) {
    int n = a.size(), m = b.size();
    for(int i = 0; i <= n; i++) pd[i][0] = i;
    for(int j = 0; j <= m; j++) pd[0][j] = j;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            int del = pd[i][j-1] + 1;
            int ins = pd[i-1][j] + 1;
            int mod = pd[i-1][j-1] + (a[i-1] != b[j-1]);
            pd[i][j] = min(del, min(ins, mod));
        }
    }
    return pd[n][m];
}
```

1.2 KMP

```
string p, t;
int b[ms], n, m;

void kmpPreprocess() {
    int i = 0, j = -1;
    b[0] = -1;
    while(i < m) {
        while(j >= 0 && p[i] != p[j]) j = b[j];
        b[++i] = ++j;
    }
}

void kmpSearch() {
    int i = 0, j = 0, ans = 0;
    while(i < n) {
        while(j >= 0 && t[i] != p[j]) j = b[j];
        i++; j++;
        if(j == m) {
            //ocorrencia aqui comecando em i - j
            ans++;
            j = borda[j];
        }
    }
}
```

1.3 Trie

```
int trie[ms][sigma], terminal[ms], z;

void init() {
    memset(trie[0], -1, sizeof trie[0]);
    z = 1;
}

int get_id(char c) {
    return c - 'a';
}

void insert(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            memset(trie[z], -1, sizeof trie[z]);
            trie[cur][id] = z++;
        }
        cur = trie[cur][id];
    }
    terminal[cur]++;
}

int count(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            return false;
        }
        cur = trie[cur][id];
    }
    return terminal[cur];
}
```

1.4 Aho-Corasick

```
// Construa a Trie do seu dicionario com o codigo acima
```

```cpp
int fail[ms], q[ms], front, rear;

void buildFailure() {
    front = 0; rear = 0; q[rear++] = 0;
    while(front < rear) {
        int node = q[front++];
        for(int pos = 0; pos < sigma; pos++) {
            int &v = trie[node][pos];
            int f = node == 0 ? 0 : trie[fail[node]][pos];
            if(v == -1) {
                v = f;
            } else {
                fail[v] = f;
                q.push(v);
                // juntar as informacoes da borda para o V ja q um match em V implica um
                      match na borda
                terminal[v] += terminal[f];
            }
        }
    }
}

int search(string &txt) {
    int node = 0;
    int ans = 0;
    for(int i = 0; i < txt.length(); i++) {
        int pos = get_id(txt[i]);
        node = trie[node][pos];
        // processar informacoes no no atual
       ans += terminal[node];
    }
    return ans;
}
```

2 Trees

2.1 BIT - Binary Indexed Tree

```cpp
int arr[ms], bit[ms], n;

void update(int v, int idx) {
    while(idx <= n) {
        bit[idx] += v;
        idx += idx & -idx;
    }
}

int query(int idx) {
    int r = 0;
    while(idx > 0) {
        r += bit[idx];
        idx -= idx & -idx;
    }
    return r;
}
```

2.2 Iterative Segment Tree

```cpp
int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void update(int p, int value) { // set value at position p
    for(t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}

int query(int l, int r) {
    int res = 0;
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if(l&1) res += t[l++];
        if(r&1) res += t[--r];
    }
    return res;
}

// If is non-commutative
S query(int l, int r) {
  S resl, resr;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) resl = combine(resl, t[l++]);
    if (r&1) resr = combine(t[--r], resr);
  }
  return combine(resl, resr);
}
```

2.3 Iterative Segment Tree with Interval Updates

```cpp
int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void update(int l, int r, int value) {
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if(l&1) t[l++] += value;
        if(r&1) t[--r] += value;
    }
}

int query(int p) {
    int res = 0;
    for(p += n; p > 0; p >>= 1) res += t[p];
    return res;
}

void push() { // push modifications to leafs
    for(int i = 1; i < n; i++) {
        t[i<<1] += t[i];
        t[i<<1|1] += t[i];
        t[i] = 0;
    }
}
```

2.4 Recursive Segment Tree

```cpp
int arr[4 * ms], seg[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}

int query(int L, int R, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(R < l || L > r) return 0;
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
}

void update(int V, int I, int idx = 0, int l = 0, int r = n -1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l > I || r < I) return;
    if(l == r) {
        arr[I] = V;
```

```
        seg[idx] = V;
        return;
    }
    update(V, I, left, l, mid); update(V, I, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}
```

2.5 Segment Tree with Lazy Propagation

```
int arr[4 * ms],  seg[4 * ms], lazy[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}

int query(int L, int R, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(lazy[idx]) {
        seg[idx] += lazy[idx] * (r - l + 1);
        if(l < r) {
            lazy[left] += lazy[idx];
            lazy[right] += lazy[idx];
        }
        lazy[idx] = 0;
    }
    if(R < l || L > r) return 0;
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
}

void update(int V, int L, int R, int idx = 0, int l = 0, int r = n -1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(lazy[idx]) {
        seg[idx] += lazy[idx] * (r - l + 1);
        if(l < r) {
            lazy[left] += lazy[idx];
            lazy[right] += lazy[idx];
        }
        lazy[idx] = 0;
    }
    if(l > R || r < L) return;
    if(L <= l && r <= R) {
        seg[idx] += V * (r - l + 1);
        if(l < r) {
            lazy[left] += V;
            lazy[right] += V;
        }
        return;
    }
    update(V, L, R, left, l, mid); update(V, L, R, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}
```

3 Graph Algorithms

3.1 Dinic Max Flow

```
const int ms = 1e3; // Quantidade maxima de vertices
const int me = 1e5; // Quantidade maxima de arestas

int adj[ms], to[me], ant[me], wt[me], z, n;
int copy_adj[ms], fila[ms], level[ms];

void clear() {
    memset(adj, -1, sizeof adj);
    z = 0;
}

int add(int u, int v, int k) {
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = k;
    adj[u] = z++;
    swap(u, v);
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = 0;
    adj[u] = z++;
}

int bfs(int source, int sink) {
        memset(level, -1, sizeof level);
        level[source] = 0;
        int front = 0, size = 0, v;
        fila[size++] = source;
        while(front < size) {
                v = fila[front++];
                for(int i = adj[v]; i != -1; i = ant[i]) {
                        if(wt[i] && level[to[i]] == -1) {
                                level[to[i]] = level[v] + 1;
                                fila[size++] = to[i];
                        }
                }
        }
        return level[sink] != -1;
}

int dfs(int v, int sink, int flow) {
        if(v == sink) return flow;
        int f;
        for(int &i = copy_adj[v]; i != -1; i = ant[i]) {
                if(wt[i] && level[to[i]] == level[v] + 1 && (f = dfs(to[i], sink, min(flow,
                    wt[i])))) {
                        wt[i] -= f;
                        wt[i ^ 1] += f;
                        return f;
                }
        }
        return 0;
}

int maxflow(int source, int sink) {
        int ret = 0, flow;
        while(bfs(source, sink)) {
                memcpy(copy_adj, adj, sizeof adj);
                while((flow = dfs(source, sink, 1 << 30))) {
                        ret += flow;
                }
        }
        return ret;
}
```

3.2 Articulations Points and Bridges

```
int idx[ms], art[ms], bridge[me], ind, child;

int dfs(int v, int par = -1) {
    int low = idx[v] = ind++;
    for(int i = adj[v]; i > -1; i = ant[i]) {
        if(idx[to[i]] == -1) {
            if(par == -1) child++;
            int temp = dfs(to[i], v);
            if(temp >= idx[v]) art[v] = true;
            if(temp > idx[v]) bridge[i] = true;
            low = min(low, temp);
        } else if(to[i] != par) low = min(low, idx[to[i]]);
    }
    return low;
}
```

```
void artPointAndBridge() {
    ind = 0;
    memset(idx, -1, sizeof idx);
    memset(art, 0, sizeof art);
    for(int i = 0; i < n; i++) if(idx[i] == -1) {
        child = 0;
        dfs(i);
        art[i] = child > 1;
    }
}
```

3.3 Biconnected Components

```
int idx[ms], bc[me], ind, nbc, child, st[me], top;

void generateBc(int edge) {
    while(st[--top] != edge) {
        bc[st[top]] = nbc;
    }
    bc[edge] = nbc++;
}

int dfs(int v, int par = -1) {
    int low = idx[v] = ind++;
    for(int i = adj[v]; i > -1; i = ant[i]) {
        if(idx[to[i]] == -1) {
            if(par == -1) child++;
            st[top++] = i;
            int temp = dfs(to[i], v);
            if(par == -1 && child > 1 || ~par && temp >= idx[v]) generateBc(i);
            if(temp >= idx[v]) art[v] = true;
            if(temp > idx[v]) bridge[i] = true;
            low = min(low, temp);
        } else if(to[i] != par && idx[to[i]] < low) {
            low = idx[to[i]];
            st[top++] = i;
        }
    }
    return low;
}

void biconnected() {
    ind = 0;
    nbc = 0;
    top = -1;
    memset(idx, -1, sizeof idx);
    for(int i = 0; i < n; i++) if(idx[i] == -1) {
        child = 0;
        dfs(i);
    }
}
```

3.4 SCC - Strongly Connected Components / 2SAT

```
int idx[ms], low[ms], ind, comp[ms], ncomp, st[ms], top;

int dfs(int v) {
    if(~idx[v]) return idx[v] ? idx[v] : ind;
    low[v] = idx[v] = idx++;
    st[top++] = v;
    onStack[v] = true;
    for(int w = adj[x]; ~w; w = ant[w]) {
        low[v] = min(low[v], dfs(to[w]));
    }
    if(low[v] == idx[v]) {
        while(top > -1) {
            int w = st[--top];
            idx[w] = 0;
            low[w] = low[v];
            comp[w] = ncomp;
        }
        ncomp++;
    }
    return low[v];
}

bool solveSat() {
    memset(idx, -1, sizeof idx);
    ind = 1; top = -1;
    for(int i = 0; i < n; i++) dfs(i);
    for(int i = 0; i < n; i++) if(low[i] == low[i^1]) return false;
    return true;
}

// Operacoes comuns de 2-sat
// ~v = "nao v"
#define trad(v) (v<0?((~v)*2)^1:v*2)
void addImp(int a, int b) { add(trad(a), trad(b)); }
void addOr(int a, int b) { addImp(~a, b); addImp(~b, a); }
void addEqual(int a, int b) { addOr(a, ~b); addOr(~a, b); }
void addDiff(int a, int b) { addEqual(a, ~b); }
// valoracao: value[v] = comp[trad(v)] < comp[trad(~v)]
```

3.5 LCA - Lowest Common Ancestor

```
int par[ms][mlg], lvl[ms];

void dfs(int v, int p, int l = 0) {
    lvl[v] = l;
    par[v][0] = p;
    for(int i = adj[v]; i > - 1; i = ant[i]) {
        if(to[i] != p) dfs(to[i], v, l + 1);
    }
}

void processAncestors(int root = 0) {
    dfs(root, root);
    for(int k = 1; k <= mlg; k++) {
        for(int i = 0; i < n; i++) {
            par[i][k] = par[par[i][k-1]][k-1];
        }
    }
}

int lca(int a, int b) {
    if(lvl[b] > lvl[a]) swap(a, b);
    for(int i = mlg; i >= 0; i--) {
        if(lvl[a] - (1 << i) >= lvl[b]) a = par[a][i];
    }
    if(a == b) return a;
    for(int i = mlg; i >= 0; i--) {
        if(par[a][i] != par[b][i]) a = par[a][i], b = par[b][i];
    }
    return par[a][0];
}
```

3.6 Floyd-Warshall / Shortest path between all pairs

```
const int inf = 0x3f3f3f3f;
int g[ms][ms], dis[ms][ms], n;

void clear() {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            g[i][j] = i == j ? 0 : inf;
        }
    }
}

void add(int u, int v, int w) {
    g[u][v] = min(w, g[u][v]);
}
```

```
void floydWarshall() {
    memcpy(g, dis, sizeof g);
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
            }
        }
    }
}
```

3.7 Disjoint-Set / Union-Find

```
int ds[ms], sz[ms], n;

void dsBuild() {
    for(int i = 0; i < n; i++) {
        ds[i] = i;
    }
}

int dsFind(int i) {
    while(ds[i] != i) {
        ds[i] = ds[ds[i]];
        i = ds[i];
    }
}

void dsUnion(int a, int b) {
    a = dsFind(a);
    b = dsFind(b);
    if(sz[a] < sz[b]) swap(a, b);
    sz[a] += sz[b];
    ds[b] = a;
}
```

3.8 Kruskal's MST - Minimum Spanning Tree

```
// Usa a estrutura de Disjoint-Set acima

typedef pair<int, int> ii;
typedef pair<int, ii> iii;

iii e[me], mst[me];
int z, mstLen;

void add(int u, int v, int w) {
    e[z++] = iii(w, ii(u, v));
}

int kruskal() {
    int ans = 0;
    dsBuild();
    sort(e, e + z);
    for(int i = 0; i < n; i++) {
        int u = e[i].second.first, v = e[i].second.second, ew = e[i].first;
        if(dsFind(u) != dsFind(v)) {
            dsUnion(u, v);
            ans += w;
            mst[mstLen++] = e[i];
        }
    }
    return ans;
}
```

4 Math

4.1 Discrete Logarithm

```
ll discreteLog(ll a, ll b, ll m) {
        // a^ans == b mod m
        // ou -1 se nao existir
        ll cur = a, on = 1;
        for(int i = 0; i < 100; i++) {
                cur = cur * a % m;
        }
        while(on * on <= m) {
                cur = cur * a % m;
                on++;
        }
        map<ll, ll> position;
        for(ll i = 0, x = 1; i * i <= m; i++) {
                position[x] = i * on;
                x = x * cur % m;
        }
        for(ll i = 0; i <= on + 20; i++) {
                if(position.count(b)) {
                        return position[b] - i;
                }
                b = b * a % m;
        }
        return -1;
}
```

4.2 GCD - Greatest Common Divisor

```
ll gcd(ll a, ll b) {
        while(b) a %= b, swap(a, b);
        return a;
}
```

4.3 Extended Euclides

```
// euclides estendido: acha u e v da equacao:
// u * x + v * y = gcd(x, y);
// u eh inverso modular de x no modulo y
// v eh inverso modular de y no modulo x

pair<ll, ll> euclides(ll a, ll b) {
        ll u = 0, oldu = 1, v = 1, oldv = 0;
        while(b) {
                ll q = a / b;
                oldv = oldv - v * q;
                oldu = oldu - u * q;
                a = a - b * q;
                swap(a, b);
                swap(u, oldu);
                swap(v, oldv);
        }
        return make_pair(oldu, oldv);
}
```

4.4 Fast Exponentiation

```
const ll mod = 1e9+7;

ll fExp(ll a, ll b) {
        ll ans = 1;
        while(b) {
```

```
		if(b & 1) ans = ans * a % mod;
		a = a * a % mod;
		b >>= 1;
	}
	return ans;
}
```

4.5 Matrix Fast Exponentiation

```
const ll mod = 1e9+7;
const int m = 2; // size of matrix

struct Matrix {
	ll mat[m][m];
	Matrix operator * (const Matrix &p) {
		Matrix ans;
		for(int i = 0; i < m; i++)
			for(int j = 0; j < m; j++)
				for(int k = ans.mat[i][j] = 0; k < m; k++)
					ans.mat[i][j] = (ans.mat[i][j] + mat[i][k] * p.mat[k
						][j]) % mod;
		return ans;
	}
};

Matrix fExp(Matrix a, ll b) {
	Matrix ans;
	for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
		ans.mat[i][j] = i == j;
	while(b) {
		if(b & 1) ans = ans * a;
		a = a * a;
		b >>= 1;
	}
	return ans;
}
```

5 Geometry

5.1 Geometry

```
const double inf = 1e100, eps = 1e-9;

struct PT {
    double x, y;
    PT(double x, double y) : x(x), y(y) {}
    PT operator + (const PT &p) { return PT(x + p.x, y + p.y); }
    PT operator - (const PT &p) { return PT(x - p.x, y - p.y); }
    PT operator * (double c) { return PT(x * c, y * c); }
    PT operator / (double c) { return PT(x / c, y / c); }
    bool operator <(const PT &p) const {
		return x < p.x || (x == p.x && y < p.y);
	}
};

double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }

// Rotaciona o ponto CCW ou CW ao redor da origem
PT rotateCCW90(PT p) { return PT(-p.y, p.x); }
PT rotateCW90(PT p) { return PT(p.y, -p.x); }
PT rotateCCW(PT p, double d) {
    return PT(p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y * cos(t));
}

// Projeta ponto c na linha a - b assumindo a != b
PT projectPointLine(PT a, PT b, PT c) {
    return a + (b - a) * dot(c - a, b - a) / dot(b - a, b - a);
}

// Projeta ponto c no segmento a - b
PT projectPointSegment(PT a, PT b, PT c) {
    double r = dot(b - a, b - a);
    if(abs(r) < eps) return a;
    r = dot(c - a, b - a) / r;
    if(r < 0) return a;
    if(r > 1) return b;
    return a + (b - a) * r;
}
// Calcula distancia entre o ponto c e o segmento a - b
double distancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, projectPointSegment(a, b, c)));
}

// Calcula distancia entre o ponto (x, y, z) e o plano ax + by + cz = d
double distancePointPlane(double x, double y, double z, double a, double b, double c, double
      d) {
    return abs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}

// Determina se as linhas a - b e c - d sao paralelas ou colineares
bool linesParallel(PT a, PT b, PT c, PT d) {
    return abs(cross(b - a, c - d)) < eps;
}

bool linesCollinear(PT a, PT b, PT c, PT d) {
    return linesParallel(a, b, c, d) && abs(cross(a - b, a - c)) < eps && abs(cross(c - d, c
          - a)) < eps;
}

// Determina se o segmento a - b intersecta com o segmento c - d
bool segmentsIntersect(PT a, PT b, PT c, PT d) {
    if(linesCollinear(a, b, c, d)) {
        if(dist2(a, c) < eps || dist2(a, d) < eps || dist2(b, c) < eps || dist2(b, d) < eps)
            return true;
        if(dot(c - a, c - b) > 0 && dot(d - a, d - b) > 0 && dot(c - b, d - b) > 0) return
            false;
        return true;
    }
    if(cross(d - a, b - a) * cross(c - a, b - a) > 0) return false;
    if(cross(a - c, d - c) * cross(b - c, d - c) > 0) return false;
    return true;
}

// Calcula a intersecao entre as linhas a - b e c - d assumindo que uma unica intersecao
      existe
// Para intersecao de segmentos, cheque primeiro se os segmentos se intersectam
PT computeLineIntersection(PT a, PT b, PT c, PT d) {
    b = b - a; d = c - d; c = c - a;
    return a + b * cross(c, d) / cross(b, d);
}

// Calcula centro do circulo dado tres pontos
PT computeCircleCenter(PT a, PT b, PT c) {
    b = (a + b) / 2;
    c = (a + c) / 2;
    return computeLineIntersection(b, b + rotateCW90(a - b), c, c + rotateCW90(a - c));
}


// Determina se o ponto esta num poligno possivelmente nao-convexo
// Retorna 1 para pontos estritamente dentro, 0 para pontos estritamente fora do poligno
// e 0 ou 1 para os pontos restantes
// Eh possivel converter num teste exato usando inteiros e tomando cuidado com a divisao
// e entao usar testes exatos para checar se esta na borda do poligno
bool pointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    if((p[i].y <= q.y && q.y < p[j].y || p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// Determina se o ponto esta na borda do poligno
bool pointOnPolygon(const vector<PT> &p, PT q) {
```

```
  for(int i = 0; i < p.size(); i++)
    if(dist2(projectPointSegment(p[i], p[(i + 1) % p.size()], q), q) < eps)
      return true;
    return false;
}

// Calcula intersecao da linha a - b com o circulo centrado em c com raio r > 0
vector<PT> circleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ans;
  b = b - a;
  a = a - c;
  double x = dot(b, b);
  double y = dot(a, b);
  double z = dot(a, a) - r * r;
  double w = y * y - x * z;
  if (w < -eps) return ans;
  ans.push_back(c + a + b * (-y + sqrt(w + eps)) / x);
  if (w > eps)
    ans.push_back(c + a + b * (-y - sqrt(w)) / x);
  return ans;
}

// Calcula intersecao do circulo centrado em a com raio r e o centrado em b com raio R
vector<PT> circleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ans;
  double d = sqrt(dist2(a, b));
  if (d > r + R || d + min(r, R) < max(r, R)) return ans;
  double x = (d * d - R * R + r * r)/(2 * d);
  double y = sqrt(r * r - x * x);
  PT v = (b - a) / d;
  ans.push_back(a + v * x + rotateCCW90(v) * y);
  if (y > 0)
    ans.push_back(a + v * x - RotateCCW90(v) * y);
  return ans;
}

// Calcula a area ou o centroide de um poligono (possivelmente nao-convexo)
// assumindo que as coordenadas estao listada em ordem horaria ou anti-horaria
// O centroide eh equivalente a o centro de massa ou centro de gravidade
double computeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    area += p[i].x * p[j].y - p[j].x * p[i].y;
  }
  return area / 2.0;
}

double computeArea(const vector<PT> &p) {
  return abs(computeSignedArea(p));
}

PT computeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for(int i = 0; i < p.size(); i++){
    int j = (i + 1) % p.size();
    c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
  }
  return c / scale;
}

// Testa se o poligno listada em ordem CW ou CCW eh simples (nenhuma linha se intersecta)
bool isSimple(const vector<PT> &p) {
  for(int i = 0; i < p.size(); i++) {
    for(int k = i + 1; k < p.size(); k++) {
      int j = (i + 1) % p.size();
      int l = (k + 1) % p.size();
      if (i == l || j == k) continue;
      if (segmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

5.2 Convex Hull

```
vector<PT> convexHull(vector<PT> p)) {
    int n = p.size(), k = 0;
    vector<PT> h(2 * n);
    sort(p.begin(), p.end());
    for(int i = 0; i < n; i++) {
        while(k >= 2 && cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]) <= 0) k--;
        h[k++] = p[i];
    }
    for(int i = n - 2, t = k + 1; i >= 0; i--) {
        while(k >= t && cross(h[k - 1] - h[k - 2], p[i] - h[k - 2]) <= 0) k--;
        h[k++] = p[i];
    }
    h.resize(k);
    return h;
}
```

5.3 ClosestPair

```
double closestPair(vector<PT> p) {
    int n = p.size(), k = 0;
    sort(p.begin(), p.end());
    double d = inf;
    set<PT> ptsInv;
    for(int i = 0; i < n; i++) {
        while(k < i && p[k].x < p[i].x - d) {
            ptsInv.erase(swapCoord(p[k++]));
        }
        for(auto it = ptsInv.lower_bound(PT(p[i].y - d, p[i].x - d));
            it != ptsInv.end() && it->x <= p[i].y + d; it++) {
            d = min(d, !(p[i] - swapCoord(*it)));
        }
        ptsInv.insert(swapCoord(p[i]));
    }
    return d;
}
```

5.4 Intersection Points

```
int intersectionPoints(vector<pair<PT, PT>> v) {
    int n = v.size();
    vector<pair<int, int>> events, vertInt;
    for(int i = 0; i < n; i++) {
        if(v.first.x == v.second.x) { // Segmento Vertical
            int y0 = min(v.first.y, v.second.y), y1 = max(v.first.y, v.second.y);
            events.push_back({v.first.x, vertInt.size()}); // Tipo = Indice no array
            vertInt.push_back({y0, y1});
        } else { // Segmento Horizontal
            int x0 = min(v.first.x, v.second.x), x1 = max(v.first.x, v.second.x);
            events.push_back({x0, -1}); // Inicio de Segmento
            events.push_back({x1, inf}); // Final de Segmento
        }
    }
    sort(events.begin(), events.end());
    int ans = 0;
    for(int i = 0; i < events.size(); i++) {
        int t = events[i].second;
        if(t == -1) {
            segUpdate(events[i].first, 1);
        } else if(t == inf) {
            segUpdate(events[i].first, 0);
        } else {
            ans += segQuery(vertInt[t].first, vertInt[t].second);
        }
    }
    return ans;
}
```

5.5 Maximum Segments Overlap (Sweep Line)

```
int maxSegOverlap(vector<pair<int, int>> v) {
    int n = v.size();
    vector<pair<int, int>> events(n * 2);
    // tipo 1 = Inicio de segmento
    // tipo 0 = Fim de segmento
    for(int i = 0; i < n; i++) {
        events[i*2] = {v[i].first, 1};
        events[i*2+1] = {v[i].second, 0};
    }
    sort(events.begin(), events.end());
    int qnt = 0, ans = 0;
    for(int i = 0; i < 2 * n; i++) {
        if(events[i].second) {
            ans = max(++qnt, ans);
        } else {
            --qnt;
        }
    }
    return ans;
}
```

6 Miscellaneous

6.1 LIS - Longest Increasing Subsequence

```
int arr[ms], lisArr[ms], n;
// int bef[ms], pos[ms];

int lis() {
    int len = 1;
    lisArr[0] = arr[0];
    // bef[0] = -1;
    for(int i = 1; i < n; i++) {
        // upper_bound se non-decreasing
        int x = lower_bound(lisArr, lisArr + len, arr[i]) - lisArr;
        len = max(len, x + 1);
        lisArr[x] = arr[i];
        // pos[x] = i;
        // bef[i] = x ? pos[x-1] : -1;
    }
    return len;
}

vi getLis() {
    int len = lis();
    vi ans;
    for(int i = pos[lisArr[len - 1]]; i >= 0; i = bef[i]) {
        ans.push_back(arr[i]);
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

6.2 Binary Search

```
int smallestSolution() {
    int x = -1;
    for(int b = z; b >= 1; b /= 2) {
        while(!ok(x+b)) x += b;
    }
    return x + 1;
}

int maximumValue() {
    int x = -1;
    for(int b = z; b >= 1; b /= 2) {
        while(f(x+b) < f(x+b+1)) x += b;
    }
    return x + 1;
}
```