Guloso Tridimensional [UFMG]

Caio Caldeira, Henrique Brito, Tomaz Gomes

Índice

1 Estruturas

1.1 SegTree

```
// Acha a soma de um segmento qualquer
// e faz o update de um elemento por vez

// Complexidade
// Build  -> O(n)
// Query  -> O(log(n))
// Update -> O(log(n))

const int MAX = (int)1e5 + 10;

namespace seg{

  int n;
  ll *vec;
  ll seg[4*MAX];

  ll make_node(ll v){
    return v;
  }
  ll make_neutro(){
    return 0;
  }
  ll combina(ll a, ll b){
    return a+b;
  }

  void build(int at=1, int l=0, int r=n-1){
    if( l==r ){
      seg[at] = make_node(vec[l]);
      return;
    }
    int m = (l+r)/2;
    build(2*at, l, m);
    build(2*at +1, m+1, r);
    seg[at] = combina(seg[2*at], seg[2*at +1]);
  }
```

```
  void build(int n_, ll *vec_){
    n = n_;
    vec = vec_;
    build();
  }

  ll query(int ql, int qr, int at=1, int l=0, int r=n-1 ){
    if( l>r || ql>r || qr<l ){
      return make_neutro();
    }
    if( ql<=l && r<=qr ){
      return seg[at];
    }
    int m = (l+r)/2;
    return combina(query(ql, qr, 2*at, l, m), query(ql, qr,
       2*at +1, m+1, r));
  }

  void update(int pos, int x, int at=1, int l=0, int r=n-1){
    if( l==r ){
      seg[at] = make_node(x);
      return;
    }
    int m = (l+r)/2;
    if( pos <= m ){
      update(pos, x, 2*at, l, m);
    }else{
      update(pos, x, 2*at +1, m+1, r);
    }
    seg[at] = combina(seg[2*at], seg[2*at +1]);
  }

};
```

1.2 SegTree 2D

```
// Consultas 0-based
// Um valor inicial em (x, y) deve ser colocado em
   seg[x+n][y+n]
// Query: soma do retangulo ((x1, y1), (x2, y2))
// Update: muda o valor da posicao (x, y) para val
// Nao pergunte como que essa coisa funciona
//
// Para query com distancia de manhattan <= d, faca
// nx = x+y, ny = x-y
// Update em (nx, ny), query em ((nx-d, ny-d), (nx+d, ny+d))
//
// Se for de min/max, pode tirar os if's da 'query', e fazer
// sempre as 4 operacoes. Fica mais rapido
//
// Complexidades:
// build - O(n^2)
// query - O(log^2(n))
// update - O(log^2(n))

int seg[2*MAX][2*MAX], n;
void build() {
    for (int x = 2*n; x; x--) for (int y = 2*n; y; y--) {
        if (x < n) seg[x][y] = seg[2*x][y] + seg[2*x+1][y];
        if (y < n) seg[x][y] = seg[x][2*y] + seg[x][2*y+1];
    }
}
int query(int x1, int y1, int x2, int y2) {
    int ret = 0, y3 = y1 + n, y4 = y2 + n;
    for (x1 += n, x2 += n; x1 <= x2; ++x1 /= 2, --x2 /= 2)
        for (y1 = y3, y2 = y4; y1 <= y2; ++y1 /= 2, --y2 /=
           2) {
            if (x1%2 == 1 and y1%2 == 1) ret += seg[x1][y1];
            if (x1%2 == 1 and y2%2 == 0) ret += seg[x1][y2];
            if (x2%2 == 0 and y1%2 == 1) ret += seg[x2][y1];
            if (x2%2 == 0 and y2%2 == 0) ret += seg[x2][y2];
        }

    return ret;
}
```

```
void update(int x, int y, int val) {
    int y2 = y += n;
    for (x += n; x; x /= 2, y = y2) {
        if (x >= n) seg[x][y] = val;
        else seg[x][y] = seg[2*x][y] + seg[2*x+1][y];

        while (y /= 2) seg[x][y] = seg[x][2*y] +
           seg[x][2*y+1];
    }
}
```

1.3 SegTree Lazy

```
// Acha a soma de um segmento qualquer
// e faz o update in range

// Complexidade
// Build  -> O(n)
// Query  -> O(log(n))
// Update -> O(log(n))

const int MAX = (int)1e5 + 10;

namespace seg{

  int n;
  int vec[MAX];
  ll seg[4*MAX], lazy[4*MAX];

  ll make_node(int v){
    return v;
  }
  ll make_neutro(){
    return 0;
  }
  ll combina(ll a, ll b){
    return a+b;
  }

  void build(int at=1, int l=0, int r=n-1){
    lazy[at]=0;
    if( l==r ){
      seg[at] = make_node(vec[l]);
      return;
    }
    int m = (l+r)/2;
    build(2*at, l, m);
    build(2*at +1, m+1, r);
    seg[at] = combina(seg[2*at], seg[2*at +1]);
  }
```

```
  void propaga(int at, int l, int r){
        seg[at] += lazy[at]*(r-l+1);
        if(l != r){
      lazy[2*at] += lazy[at];
      lazy[2*at+1] += lazy[at];
    }
        lazy[at] = 0;
  }

  ll query(int ql, int qr, int at=1, int l=0, int r=n-1 ){
    propaga(at, l, r);
    if( l>r || ql>r || qr<l ){
      return make_neutro();
    }
    if( ql<=l && r<=qr ){
      return seg[at];
    }
    int m = (l+r)/2;
    return combina(query(ql, qr, 2*at, l, m), query(ql, qr,
       2*at +1, m+1, r));
  }

  void update(int ul, int ur, int x, int at=1, int l=0, int
     r=n-1){
    propaga(at, l, r);
    if( l>r || ul>r || ur<l ){
      return;
    }
    if( ul<=l and r<=ur ){
      lazy[at] += x;
      propaga(at, l, r);
      return;
    }
    int m = (l+r)/2;
    update(ul, ur, x, 2*at, l, m);
    update(ul, ur, x, 2*at +1, m+1, r);
    seg[at] = combina(seg[2*at], seg[2*at +1]);
  }
}
```

1.4 SegTree Iterativa

```
// Consultas 0-based
// Valores iniciais devem estar em (seg[n], ... , seg[2*n-1])
// Query: soma do range [a, b]
// Update: muda o valor da posicao p para x
//
// Complexidades:
// build - O(n)
// query - O(log(n))
// update - O(log(n))

int seg[2 * MAX];
int n;

void build() {
    for (int i = n - 1; i; i--) seg[i] = seg[2*i] +
       seg[2*i+1];
}

int query(int a, int b) {
    int ret = 0;
    for(a += n, b += n; a <= b; ++a /= 2, --b /= 2) {
        if (a % 2 == 1) ret += seg[a];
        if (b % 2 == 0) ret += seg[b];
    }
    return ret;
}

void update(int p, int x) {
    seg[p += n] = x;
    while (p /= 2) seg[p] = seg[2*p] + seg[2*p+1];
}
```

1.5 Merge Sort Tree

```
// Quantos n meros <= k existem em um intervalo

// Complexidade
// Espacial - O(nlog(n))
// Build - O(nlog(n))
// Query - O(log(n)^2)

const int MAX = (int)1e5;

int n, k;
vector<int> tree[4*MAX];
int v[MAX];

void merge( vector<int>& v1, vector<int>& v2, vector<int>&
    ret ){
     int i=0;
     int j=0;

     while( i<v1.size() and j<v2.size() ){
          if( v1[i] < v2[j] ){
               ret.pb(v1.at(i));
               i++;
          }else{
               ret.pb(v2.at(j));
               j++;
          }
     }
     while( i<v1.size() ){
          ret.pb(v1.at(i));
          i++;
     }
     while( j<v2.size() ){
          ret.pb(v2.at(j));
          j++;
     }

}

void build(int at=1, int b=0, int e=n-1){
  if( b==e ){
        tree[at].pb(v[b]);
      return;
  }
  int m = (b+e)/2;
  build(2*at, b, m);
  build(2*at +1, m+1, e);
  merge(tree[2*at], tree[2*at +1], tree[at]);
}

int query(int qb, int qe, int at=1, int b=0, int e=n-1){
  if( b>e || qb>e || qe<b ){
    return 0;
  }
  if( qb<=b && e<=qe ){
        return upper_bound(tree[at].begin(), tree[at].end(),
           k)-tree[at].begin();
  }
  int m = (b+e)/2;
  return query(qb, qe, 2*at, b, m) + query(qb, qe, 2*at +1,
     m+1, e);
}
```

1.6 DSU

```
// Complexidade
// Build - O(n)
// Find - O(1)
// Union - O(1)

const ll MAX = (ll)3e5 + 10;

int n, id[MAX], size[MAX];

void build(){
    for( int i=0; i<n; i++ ){
        id[i] = i;
        size[i] = 1;
    }
}

int find(int k){
    return id[k] == k ? k : id[k] = find(id[k]);
}

void unite(int a, int b){
    a = find(a);
    b = find(b);
    if( size[a] > size[b] ){
        swap(a, b);
    }
    id[a] = b;
    size[b] += size[a];
}
```

1.7 DSU Persistente

```
// Union-find com union by rank mais 2 fun   es

// Union e find_current
//   Complexidade - O(log(n))

// Find_time -> executa o find depois no tempo t
//   Complexidade - O(log(n))

// Roll_back -> retorna ao estado anterior
//   Complexidade - O(1)

struct persistent_dsu{

  int dsu_size, tempo, num_comp;
  vector<int> comp_size;
  vector<pii> id;
  stack<int> stk;

  persistent_dsu(int in){
    num_comp = dsu_size = in;
    tempo = 0;
    id.resize(dsu_size);
    comp_size.resize(dsu_size);
    for( int i=0; i<dsu_size; i++ ){
      id[i] = {i, tempo};
      comp_size[i] = 1;
    }
  }

    persistent_dsu() : persistent_dsu(10){}

  int find_current(int k){
    return id[k].f == k ? k : find_current(id[k].f);
  }

  int find_time(int k, int t){
    if( t < id[k].s ) return k;
    return id[k].f == k ? k : find_time(id[k].f, t);
  }
```

```
  void roll_back(){
    if( stk.empty() ) return;
    int u = stk.top(); stk.pop();
    if( id[u].f == u ) return;
        id[u] = {u, --tempo};
  }

  void unite(int a, int b){
    a = find_current(a);
    b = find_current(b);
    if( comp_size[a] > comp_size[b] ){
        swap(a, b);
    }
        if( a == b ){
            tempo++;
            stk.push(a);
            return;
        }
        num_comp--;
    id[a] = {b, ++tempo};
    comp_size[b] += comp_size[a];
    stk.push(a);
  }

    int size(){
        return num_comp;
    }

};
```

1.8 Prefix Matrix

```
// Acha a soma dos elementos de um sub-ret ngulo da matriz

// Matriz identada em 1

// Complexidade
// Build - O(n^2)
// Query - O(1)

#define f first
#define s second
typedef pair<int,int> pii;

const int MAX = (int)1e3+10;

int n;
int m[MAX][MAX];
int pre[MAX][MAX];

void build(){
  for(int i=0; i<=n; i++ ){
      pre[i][0] = pre[0][i] = 0;
  }
  for( int i=1; i<=n; i++ ){
    for( int j=1; j<=n; j++ ){
      pre[i][j] = pre[i-1][j] + pre[i][j-1] - pre[i-1][j-1]
          + m[i][j];
    }
  }
}

int query( pii a, pii b ){
  return pre[b.f][b.s] - pre[b.f][a.s-1] - pre[a.f-1][b.s] +
     pre[a.f-1][a.s-1];
}
```

1.9 Order Statistic Set

```
// Funciona do C++11 pra cima
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <class T>
using ord_set = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
// para declarar:
ord_set<int> s;
// coisas do set normal funcionam:
for (auto i : s) cout << i << endl;
cout << s.size() << endl;
// k-esimo maior elemento O(log|s|):
// k=0: menor elemento
cout << *s.find_by_order(k) << endl;
// quantos sao menores do que k O(log|s|):
cout << s.order_of_key(k) << endl;
// Para fazer um multiset, tem que
// usar ord_set<pair<int, int> > com o
// segundo parametro sendo algo para diferenciar
// os ementos iguais.
// s.order_of_key({k, -INF}) vai retornar o
// numero de elementos < k
```

1.10 HLD Aresta

```
// query / update de soma das arestas
//
// Complexidades:
// build - O(n)
// query_path - O(log^2 (n))
// update_path - O(log^2 (n))
// query_subtree - O(log(n))
// update_subtree - O(log(n))

#define f first
#define s second

namespace seg {
    ll seg[4*MAX], lazy[4*MAX];
    int n, *v;
    ll build(int p=1, int l=0, int r=n-1) {
        lazy[p] = 0;
        if (l == r) return seg[p] = v[l];
        int m = (l+r)/2;
        return seg[p] = build(2*p, l, m) + build(2*p+1, m+1,
            r);
    }
    void build(int n2, int* v2) {
        n = n2, v = v2;
        build();
    }
    void prop(int p, int l, int r) {
        seg[p] += lazy[p]*(r-l+1);
        if (l != r) lazy[2*p] += lazy[p], lazy[2*p+1] +=
            lazy[p];
        lazy[p] = 0;
    }
    ll query(int a, int b, int p=1, int l=0, int r=n-1) {
        prop(p, l, r);
        if (a <= l and r <= b) return seg[p];
        if (b < l or r < a) return 0;
        int m = (l+r)/2;
        return query(a, b, 2*p, l, m) + query(a, b, 2*p+1,
            m+1, r);
    }
```

```
    ll update(int a, int b, int x, int p=1, int l=0, int
       r=n-1) {
        prop(p, l, r);
        if (a <= l and r <= b) {
            lazy[p] += x;
            prop(p, l, r);
            return seg[p];
        }
        if (b < l or r < a) return seg[p];
        int m = (l+r)/2;
        return seg[p] = update(a, b, x, 2*p, l, m) +
            update(a, b, x, 2*p+1, m+1, r);
    }
};

namespace hld {
    vector<pair<int, int> > g[MAX];
    int in[MAX], out[MAX], sz[MAX];
    int sobe[MAX], pai[MAX];
    int h[MAX], v[MAX], t;

    void build_hld(int k, int p = -1, int f = 1) {
        v[in[k] = t++] = sobe[k]; sz[k] = 1;
        for (auto& i : g[k]) if (i.f != p) {
            sobe[i.f] = i.s; pai[i.f] = k;
            h[i.f] = (i == g[k][0] ? h[k] : i.f);
            build_hld(i.f, k, f); sz[k] += sz[i.f];

            if (sz[i.f] > sz[g[k][0].f]) swap(i, g[k][0]);
        }
        out[k] = t;
        if (p*f == -1) build_hld(h[k] = k, -1, t = 0);
    }
    void build(int root = 0) {
        t = 0;
        build_hld(root);
        seg::build(t, v);
    }

    ll query_path(int a, int b) {
        if (a == b) return 0;
        if (in[a] < in[b]) swap(a, b);

        if (h[a] == h[b]) return seg::query(in[b]+1, in[a]);
        return seg::query(in[h[a]], in[a]) +
            query_path(pai[h[a]], b);
    }
    void update_path(int a, int b, int x) {
        if (a == b) return;
        if (in[a] < in[b]) swap(a, b);

        if (h[a] == h[b]) return (void)seg::update(in[b]+1,
            in[a], x);
        seg::update(in[h[a]], in[a], x);
            update_path(pai[h[a]], b, x);
    }
    ll query_subtree(int a) {
        if (in[a] == out[a]-1) return 0;
        return seg::query(in[a]+1, out[a]-1);
    }
    void update_subtree(int a, int x) {
        if (in[a] == out[a]-1) return;
        seg::update(in[a]+1, out[a]-1, x);
    }
    int lca(int a, int b) {
        if (in[a] < in[b]) swap(a, b);
        return h[a] == h[b] ? b : lca(pai[h[a]], b);
    }
};
```

1.11 HLD Vértice

```
//
// SegTree de soma
// query / update de soma dos vertices
//
// Complexidades:
// build - O(n)
// query_path - O(log^2 (n))
// update_path - O(log^2 (n))
// query_subtree - O(log(n))
// update_subtree - O(log(n))

namespace seg {
    ll seg[4*MAX], lazy[4*MAX];
    int n, *v;

    ll build(int p=1, int l=0, int r=n-1) {
        lazy[p] = 0;
        if (l == r) return seg[p] = v[l];
        int m = (l+r)/2;
        return seg[p] = build(2*p, l, m) + build(2*p+1, m+1,
            r);
    }
    void build(int n2, int* v2) {
        n = n2, v = v2;
        build();
    }
    void prop(int p, int l, int r) {
        seg[p] += lazy[p]*(r-l+1);
        if (l != r) lazy[2*p] += lazy[p], lazy[2*p+1] +=
            lazy[p];
        lazy[p] = 0;
    }
    ll query(int a, int b, int p=1, int l=0, int r=n-1) {
        prop(p, l, r);
        if (a <= l and r <= b) return seg[p];
        if (b < l or r < a) return 0;
        int m = (l+r)/2;
        return query(a, b, 2*p, l, m) + query(a, b, 2*p+1,
            m+1, r);
    }
    ll update(int a, int b, int x, int p=1, int l=0, int
        r=n-1) {
        prop(p, l, r);
        if (a <= l and r <= b) {
            lazy[p] += x;
            prop(p, l, r);
            return seg[p];
        }
        if (b < l or r < a) return seg[p];
        int m = (l+r)/2;
        return seg[p] = update(a, b, x, 2*p, l, m) +
            update(a, b, x, 2*p+1, m+1, r);
    }
};

namespace hld {
    vector<int> g[MAX];
    int in[MAX], out[MAX], sz[MAX];
    int peso[MAX], pai[MAX];
    int h[MAX], v[MAX], t;

    void build_hld(int k, int p = -1, int f = 1) {
        v[in[k] = t++] = peso[k]; sz[k] = 1;
        for (auto& i : g[k]) if (i != p) {
            pai[i] = k;
            h[i] = (i == g[k][0] ? h[k] : i);
            build_hld(i, k, f); sz[k] += sz[i];

            if (sz[i] > sz[g[k][0]]) swap(i, g[k][0]);
        }
        out[k] = t;
        if (p*f == -1) build_hld(h[k] = k, -1, t = 0);
    }
    void build(int root = 0) {
        t = 0;
        build_hld(root);
        seg::build(t, v);
    }
    ll query_path(int a, int b) {
        if (a == b) return seg::query(in[a], in[a]);
        if (in[a] < in[b]) swap(a, b);
```

```
        if (h[a] == h[b]) return seg::query(in[b], in[a]);
        return seg::query(in[h[a]], in[a]) +
            query_path(pai[h[a]], b);
    }
    void update_path(int a, int b, int x) {
        if (a == b) return (void)seg::update(in[a], in[a],
            x);
        if (in[a] < in[b]) swap(a, b);

        if (h[a] == h[b]) return (void)seg::update(in[b],
            in[a], x);
        seg::update(in[h[a]], in[a], x);
            update_path(pai[h[a]], b, x);
    }
    ll query_subtree(int a) {
        if (in[a] == out[a]-1) return seg::query(in[a],
            in[a]);
        return seg::query(in[a], out[a]-1);
    }
    void update_subtree(int a, int x) {
        if (in[a] == out[a]-1) return
            (void)seg::update(in[a], in[a], x);
        seg::update(in[a], out[a]-1, x);
    }
    int lca(int a, int b) {
        if (in[a] < in[b]) swap(a, b);
        return h[a] == h[b] ? b : lca(pai[h[a]], b);
    }
};
```

2 Grafos

2.1 Articulation Points

```
// Complexidade - O( n+m )

int n, m, timer=0;
vector<int> g[MAX];
bool vist[MAX];
int tin[MAX], low[MAX];
set<int> cut;

void dfs( int u=0, int p=-1 ){
  vist[u] = true;
  tin[u] = low[u] = timer++;
  int child=0;
  for( int e : g[u] ) if( e!=p ){
    if( vist[e] ){
      low[u] = min(low[u], tin[e]);
    }else{
      dfs(e, u);
      low[u] = min(low[u], low[e]);
      if( low[e] >= tin[u] and p!=-1 ){
        cut.insert(u);
      }
      child++;
    }
  }
  if( p==-1 and child>1 ){
    cut.insert(u);
  }
}
```

2.2 Bridges

```
// Complexidade - O(n + m)

int n, timer=0;
vector<int> g[MAX];
int tin[MAX], low[MAX];
bool vist[MAX];
vector<pair<int, int> > bridges;

void dfs( int u=0, int p=0 ){
  vist[u] = true;
  low[u] = tin[u] = timer++;
  for( int e : g[u] ) if( e!=p ){
    if( vist[e] ){
      low[u] = min(low[u], tin[e]);
    }else{
      dfs(e, u);
      low[u] = min(low[u], low[e]);
      if( low[e] > tin[u] ){
        bridges.pb({u, e});
      }
    }
  }
}
```

2.3 Centroid

```
// Complexidade - O(n*log(n))

const int MAX = (int)1e5+10;
int n;
vector<int> g[MAX];
int pai[MAX];
int sizet[MAX];
bool rev[MAX];
int lvl[MAX];

int centroid( int u, int p, int size ){
  for( int e : g[u] ) if( e!=p and !rev[e] ){
    if( sizet[e] > size/2 ){
      return centroid(e, u, size);
    }
  }
  return u;
}
int get_size( int u, int p ){
  sizet[u] = 1;
  for( int e : g[u] ) if( e!=p and !rev[e] ){
    sizet[u] += get_size(e, u);
  }
  return sizet[u];
}
void decomp( int u, int p ){
  get_size(u, u);
  int c = centroid(u, u, sizet[u]);
  rev[c] = true;
  if( u==p ){
    pai[c] = c;
  }else{
    pai[c] = p;
  }
  for( int e : g[c] ) if( !rev[e] ){
    decomp(e, c);
  }
}
```

```
int get_lvl( int u ){
  if( lvl[u] != -1 ){
    return lvl[u];
  }
  if( pai[u] == u ){
    return lvl[u] = 0;
  }
  return lvl[u] = get_lvl(pai[u])+1;
}

void build_centroid(){
  for( int i=0; i<n; i++ ){
    rev[i] = 0;
    lvl[i] = -1;
  }
  decomp(0, 0);
  for( int i=0; i<n; i++ ){
    get_lvl(i);
  }
}
```

2.4 Dijkstra

```
//  Acha a menor distancia entre um vertice ate todos os
    outros

//  Complexidade - O(m log(n))

#define mp make_pair
#define f first
#define s second
const int INF = 0x3f3f3f3f;
const int MAX = (int)1e5+10;

vector<pair< int, int > > g[MAX];
int n, d[MAX];

void dijkstra( int v ){
  for( int i=0; i<n; i++ ){
    d[i] = INF;
  }
  d[v] = 0;

  priority_queue<pair<int, int> > q;
  q.push(mp(0, v));

  while( !q.empty() ){
    int u = q.top().s;
    int dist = -q.top().f;
    q.pop();

    if( dist > d[u] )continue;
    for( auto e : g[u] ){
      if( d[e.f] > d[u] + e.s ){
        d[e.f] = d[u] + e.s;
        q.push(mp(-d[e.f], e.f));
      }
    }
  }
}
```

2.5 Dinic

```
// Acha o max flow/ min cut entre 2 vertices

// Complexidade - O(n^2*m)

struct Edge{
    int v, rev;
    ll cap;
    Edge(int v_, ll cap_, int rev_) : v(v_), rev(rev_),
       cap(cap_) {}
};

struct Dinic{
    vector<vector<Edge> > g;
    vector<int> level;
    queue<int> q;
    ll flow;
    int n;

    Dinic(int n_) : g(n_), level(n_), n(n_) {}

    void addEdge(int u, int v, ll cap){
        if(u == v) return;
        Edge e(v, cap, int(g[v].size()));
        Edge r(u, 0, int(g[u].size()));
        g[u].push_back(e);
        g[v].push_back(r);
    }

    bool buildLevelGraph(int src, int sink){
        fill(level.begin(), level.end(), -1);
        while(not q.empty()) q.pop();
        level[src] = 0;
        q.push(src);
        while(not q.empty())
        {
            int u = q.front();
            q.pop();
            for(vector<Edge>::iterator
               e=g[u].begin();e!=g[u].end();e++)
            {
                if(not e->cap or level[e->v] != -1) continue;
                level[e->v] = level[u] + 1;
                if(e->v == sink) return true;
                q.push(e->v);
            }
        }
        return false;
    }

    ll blockingFlow(int u, int sink, ll f){
        if(u == sink or not f) return f;
        ll fu = f;
        for(vector<Edge>::iterator
           e=g[u].begin();e!=g[u].end();e++)
        {
            if(not e->cap or level[e->v] != level[u] + 1)
               continue;
            ll mincap = blockingFlow(e->v, sink, min(fu,
               e->cap));
            if(mincap)
            {
                g[e->v][e->rev].cap += mincap;
                e->cap -= mincap;
                fu -= mincap;
            }
        }
        if(f == fu) level[u] = -1;
        return f - fu;
    }

    ll maxFlow(int src, int sink){
        flow = 0;
        while(buildLevelGraph(src, sink))
            flow+= blockingFlow(src, sink,
               numeric_limits<ll>::max());
        return flow;
    }
};
```

2.6 Floyd-Warshall

```
// Calcula o caminho m nimo para todos os pares de
    v rtices em um grafo n o direcionado

// Complexidade - O(n^3)

// Antes da k- sima  itera  o , j  foi calculado o
// caminho min mo de todos os pares utilizando apenas
// os v rtices {1, 2, ..., k-1} como v rtices internos

for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

2.7 Kruskal

```
// Usando Union Find com path compression em O(log(n))

// Complexidade - O(m*(log(n)+log(m))

#define mp make_pair
#define pb push_back
#define f first
#define s second
const ll MAX = (ll)3e5 + 10;

vector<pair<int,pair<int,int> > > art;
vector<bool> mst;
int n, m, id[MAX];

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

int find(int k){
    return id[k] == k ? k : id[k] = find(id[k]);
}

void unite(int a, int b){
    id[find(a)] = find(b);
}

void kruskal(){
    build();
    sort(art.begin(), art.end());
    for( auto e : art ){
        if( find(e.s.f) != find(e.s.s) ){
            unite(e.s.f, e.s.s);
            mst.pb(1);
        }else{
            mst.pb(0);
        }
    }
}
```

2.8 LCA

```
//  Usando Binary Lifting

//  Complexidade
//    Build - O(n*log(n))
//    Query - O(log(n))

const int LOG = (int)32;
const int MAX = (int)1e3+10;

int n, t=0;
int in[MAX], out[MAX];
int dp[MAX][LOG];
vector<int> g[MAX];

void dfs( int u, int p ){
  in[u] = t++;
  for( int e : g[u] ) if( e != p ){
    dp[e][0] = u;
    dfs(e, u);
  }
  out[u] = t;
}

void build(){
  for( int i=0; i<n; i++ ){
    dp[i][0] = i;
  }
  t = 0;
  dfs(0, 0);
  for( int k=1; k<LOG; k++ ){
    for( int i=0; i<n; i++ ){
      dp[i][k] = dp[dp[i][k-1]][k-1];
    }
  }
}

bool anc( int p, int f ){
  return in[p] <= in[f] and out[f] <= out[p];
}

int lca( int u, int v ){
  if( anc(u, v) ){
    return u;
  }
  if( anc(v, u) ){
    return v;
  }
  for( int k=LOG-1; ~k; k-- ){
    if( !anc(dp[u][k], v) ){
      u = dp[u][k];
    }
  }
  return dp[u][0];
}
```

2.9 Tarjan

```
// Acha todos os componentes fortemente conexos de um grafo

// Complexidade - O(n+m)

int n, m, p=0;
vector<int> g[MAX];
stack<int> s;
vector<int> visited(MAX, 0);
int id[MAX], comp[MAX];

int tarjan( int v ){
  int low = p++;
  id[v] = low;
  visited[v] = 1;

  s.push(v);
  for( int e : g[v] ){
    if( !visited[e] ){
      low = min(low, tarjan(e));
    }else if( visited[e] == 1 ){
      low = min(low, id[e]);
    }
  }

  if( low == id[v] ){
    while(1){
      int u = s.top();
      s.pop();
      visited[u] = 2;
      comp[u] = v;
      if( u == v ){
        break;
      }
    }
  }

  return low;
}
```

2.10 Topo Sort

```
// Ordem dos vertices em um DAG

// Complexidade - O(n+m)

int n;
vector<int> g[MAX];
vector<bool> vis(MAX, 0);
vector<int> ts;

void dfs( int at ){
    vis[at] = 1;
    for( int e : g[at] ) if( !vis[e] ){
        dfs(e);
    }
    ts.pb(at);
}

void topo_sort(){
    for( int i=0; i<n; i++ ){
        if( !vis[i] ){
            dfs(i);
        }
    }
    reverse(ts.begin(), ts.end());
}
```

3 Matemática

3.1 Crivo de Erastónes

```
// Acha todos os primos ate certo numero (lim)

// Complexidade - O(n*log(log(n)))

int lim;
vector<bool> is_prime;

void crivo(){
  is_prime.resize(lim+1, 1);
  is_prime[0] = is_prime[1] = 0;

  for(int i=2; i<=lim; i++){
    if(is_prime[i]){
      for(int j=i*i; j<=lim; j+=i){
        is_prime[j] = 0;
      }
    }
  }

}
```

3.2 Exponenciação Rápida

```
// (A^B) % MOD

// Complexidade - O(log(b))

typedef long long ll;

ll binpow(ll a, ll b, ll mod){
  a %= mod;
  ll ret = 1;
  while(b > 0){
    if(b & 1){
      ret = (ret*a) % mod;
    }
    a = (a*a) % mod;
    b = b >> 1;
  }
  return ret;
}
```

3.3 MDC

```
int mdc(int a, int b){
  if( !b ){
    return a;
  }else{
    return mdc(b, a%b);
  }
}

int mdc(int a, int b){
  while( b ){
    a = a%b;
    int aux = a;
    a = b;
    b = aux;
  }
  return b;
}
```

3.4 MDC Extendido

```
// Acha o mdc de a e b (retorno da fun   o)
// e os coeficientes x e y tais que:
// a*x + b*y = mdc(a, b)

int mdc_ext(int a, int b, int &x, int &y){
  int x1, y1;
  int d;
  if( a==0 ){
    x = 0;
    y = 1;
    return b;
  }
  d = mdc_ext(b%a, a, x1, y1);
  x = y1 - (b/a) *x1;
  y = x1;
  return d;
}

int inv_mod(int a, int b){
    return 1<a ? b - inv(b%a,a)*b/a : 1;
}
```

3.5 Miller Rabin

```
// Complexidade - O(log(n))

ll mul(ll x, ll y, ll mod) {
    if (!y) return 0;

    ll ret = mul(x, y >> 1, mod);
    ret = (ret + ret) % mod;
    if (y & 1) ret = (ret + x) % mod;
    return ret;
}

ll binpow(ll a, ll b, ll mod){
  a %= mod;
  ll ret = 1;
  while(b > 0){
    if(b & 1){
      ret = mul(ret, a, mod);
    }
    a = mul(a, a, mod);
    b = b >> 1;
  }
  return ret;
}

bool composite( ll n, int a, ll d, int r ){
  ll x = binpow(a, d, n);
  if( x == 1 or x == n-1 ){
    return false;
  }
  for( int i=1; i<r; i++ ){
    x = mul(x, x, n);
    if( x == n-1 ){
      return false;
    }
  }
  return true;
}

bool miller_rabin(ll n){

  if( n <= 1 ){
    return false;
  }
  if( n == 2 or n == 3 ){
    return true;
  }
  if( (n&1) == 0 ){
    return false;
  }

  int r=0;
  ll d = n-1;
  while( (n&1) == 0 ){
    d = d >> 1;
  }

  for( int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}
     ){
    if( n == a ){
      return true;
    }else if( composite(n, a, d, r) ){
      return false;
    }
  }
  return true;
}
```

3.6 Pollards Rho

```
// Fatora um numero
// Fatores nao estao ordenados

// Complexidade - O(rapidao)

mt19937 rng((int)
    chrono::steady_clock::now().time_since_epoch().count());

ll mul(ll x, ll y, ll mod) {
    if (!y) return 0;

    ll ret = mul(x, y >> 1, mod);
    ret = (ret + ret) % mod;
    if (y & 1) ret = (ret + x) % mod;
    return ret;
}


ll binpow(ll a, ll b, ll mod){
  a %= mod;
  ll ret = 1;
  while(b > 0){
    if(b & 1){
      ret = mul(ret, a, mod);
    }
    a = mul(a, a, mod);
    b = b >> 1;
  }
  return ret;
}


bool composite( ll n, int a, ll d, int r ){
  ll x = binpow(a, d, n);
  if( x == 1 or x == n-1 ){
    return false;
  }
  for( int i=1; i<r; i++ ){
    x = mul(x, x, n);
    if( x == n-1 ){
      return false;
    }
  }
  return true;
}

bool miller_rabin(ll n){

  if( n <= 1 ){
    return false;
  }
  if( n == 2 or n == 3 ){
    return true;
  }
  if( (n&1) == 0 ){
    return false;
  }

  int r=0;
  ll d = n-1;
  while( (n&1) == 0 ){
    d = d >> 1;
  }

  for( int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}
     ){
    if( n == a ){
      return true;
    }else if( composite(n, a, d, r) ){
      return false;
    }
  }
  return true;
}

ll rho(ll n) {
    if (n == 1 || miller_rabin(n)) return n;
    if (n % 2 == 0) return 2;

    while (1) {
        ll x = 2, y = 2;
```

```
        ll ciclo = 2, i = 0;

        ll c = (rng() / (double) RAND_MAX) * (n - 1) + 1;
        ll d = 1;

        while (d == 1) {
            if (++i == ciclo) ciclo *= 2, y = x;
            x = (mul(x, x, n) + c) % n;

            if (x == y) break;

            d = __gcd(abs(x - y), n);
        }

        if (x != y) return d;
    }
}

void fact(ll n, vector<ll>& v) {
    if (n == 1) return;
    if (miller_rabin(n)){
    v.pb(n);
  }
    else {
        ll d = rho(n);
        fact(d, v);
        fact(n / d, v);
    }
}
```

3.7 Totiente

```
// Usando fatora  o

// Complexidade - O(sqrt(n))

ll phi(ll n){
  ll result = n;
  for( ll i=2; i*i<=n; i++ ){
    if( n%i == 0 ){
      while( n%i == 0 ){
        n /= i;
      }
      result -= result/i;
    }
  }
  if( n>1 ){
    result -= result/n;
  }
  return result;
}
```

4 Problemas

4.1 2-SAT

```
// Complexidade - O(n+m)

// 2k - Vari vel normal
// 2k +1 - Vari vel negada

int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
    used[v] = true;
    for (int u : g[v]) {
        if (!used[u])
            dfs1(u);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : gt[v]) {
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}

bool solve_2SAT() {
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        if (!used[i])
            dfs1(i);
    }

    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
}
```

4.2 LIS

```
vector<int> v;

template<typename T> int lis(vector<T> &v){
    vector<T> ans;
    for (T t : v){
        auto it = lower_bound(ans.begin(), ans.end(), t);
        if (it == ans.end()) ans.push_back(t);
        else *it = t;
    }
    return ans.size();
}

//LIS com duplicata

vector<int> v;

template<typename T> int lis(vector<T> &v){
    vector<T> ans;
    for (T t : v){
        auto it = upper_bound(ans.begin(), ans.end(), t);
        if (it == ans.end()) ans.push_back(t);
        else *it = t;
    }
    return ans.size();
}
```

5 String

5.1 Aho Corasick

```
#include <queue>
#include <set>
#include <map>

using namespace std;

class AhoCorasick{

public:
    map< int, string > st_w;
    vector< vector< int > > G, Aut;
    vector< int > F, N;
    vector< bool > endOfWord;
    vector< string > Dic;
    int maxc, s_alf, prx;

    void insert(string &S){
        int at = 0;

        for(char c : S){
            int let = c - 'a';
            if(G[at][let] == -1) G[at][let] = prx++;
            at = G[at][let];
        }

        endOfWord[at] = true;
        st_w[at] = S;
    }

    void search(string &T){
        int at = 0;
        for(int i = 0; i<T.size(); i++){
            int let = T[i] - 'a'; // letra base do alfabeto
            at = Aut[at][let];
            if(endOfWord[at]){
                cout << "Word found! Position : " << i <<
                    '\n';
```

```
            cout << st_w[at] << '\n';
        }

        int state = at;
        while(N[state] != 0){
            state = N[state];
            cout << st_w[state] << '\n';
        }
    }
}

void makelink(const tuple<int, int, int> &p){
    int state = get<0>(p), par = get<1>(p), cpar =
       get<2>(p);
    if(par == 0){
        F[state] = 0;
        N[state] = 0;
    }
    else{
        int pre = F[par];
        while(G[pre][cpar] == -1) pre = F[pre];
        F[state] = G[pre][cpar];

        if(endOfWord[F[state]]) N[state] = F[state];
        else N[state] = N[F[state]];
    }
}

AhoCorasick(vector< string > &Dic_, int maxc_ = 1e6, int
   s_alf_ = 26){
    maxc = maxc_;
    s_alf = s_alf_;
    Dic = Dic_;
    prx = 1;

    G = vector< vector< int > >(maxc, vector<int>(s_alf,
       -1));
    Aut = vector < vector< int > >(maxc,
       vector<int>(s_alf));
    F = vector< int >(maxc);
    N = vector< int >(maxc);
    endOfWord = vector< bool >(maxc, false);

        queue< tuple<int, int , int > > q;

        for(int i = 0; i<Dic.size(); i++) insert(Dic[i]);
        for(int i = 0; i<s_alf; i++){
            if(G[0][i] == -1)  G[0][i] = 0;
            else q.push({G[0][i], 0, i});

            Aut[0][i] = G[0][i];
        }

        while(!q.empty()){
            tuple<int, int, int> p = q.front(); q.pop();

            makelink(p);

            int state = get<0>(p);

            for(int i = 0; i<s_alf; i++){
                if(G[state][i] != -1){
                    q.push({G[state][i], state, i});
                    Aut[state][i] = G[state][i];
                }
                else{
                    Aut[state][i] = Aut[F[state]][i];
                }
            }
        }
    }

};

int main(){
    vector<string> Dic = {"hers", "she", "his", "he", "to"};

    AhoCorasick AC(Dic, 100, 26);
    string s = "sheshehersahsheahtoototo";
    AC.search(s);

    return 0;
}
```

5.2 Hashing

```
//   Complexidade
// Build - O(|s|)
// Get_hash - O(1)
//
//   P e Mod
// Sao primos positivos
//
//   P deve ser parecido ao numero de caracteres
// So letra minuscula -> 31
// Maiuscula e minuscula -> 53
// Toda a ASCII -> 257
//
//   Mod deve ser grande
// 1e9+7 ou 1e9+9
//
// Comparar somente strings do mesmo tamanho para evitar
   colisao
// Se continuar com colisao, fazer 2 hashs
// Probabilidade de colisao - 1/Mod

typedef long long ll;

ll h[MAX], pwr[MAX];
const ll p = 31, mod = 1e9+7;
int n; string s;

void build(){
  pwr[0] = 1;
  for( int i=1; i<n; i++ ){
    pwr[i] = pwr[i-1]*p % mod;
  }
  h[0] = s[0];
  for( int i=1; i<n; i++ ){
    h[i] = (h[i-1]*p + s[i]) % mod;
  }
}

ll get_hash(int i, int j){
  if ( i == 0 ){
    return h[j];
  }
  return (h[j] - h[i-1]*pwr[j-i+1] % mod + mod) % mod;
}
```

5.3 KMP

```
//O(|Ptt|)
void buildKMP(string Ptt){
    lps.resize(Ptt.size());
    lps[0] = 0;
    int i = 1, j = 0;
    while(i < Ptt.size()){ //preenche lps[i] (ja tem todos
       ate i-1 calculados)
        if(Ptt[i] == Ptt[j]){
            i++;
            j++;
            lps[i-1] = j;
        }
        else{
            if(j == 0){
                lps[i] = 0;
                i++;
            }
            else j = lps[j-1];
        }
    }
}

//O(|Txt|)
void searchKMP(string Ptt, string Txt){
    int i = 0, j = 0, n = Txt.size(), m = Ptt.size();
    while(i - j <= n - m){
        if(Ptt[j] == Txt[i]){
            i++;
            j++;
            if(j == m){
                cout << "achei: " << i-j << '\n';
                j = lps[j-1];
            }
        }
        else{
            if(j == 0) i++;
            else j = lps[j-1];
        }
    }
}
```

5.4 Trie

```
#include <bits/stdc++.h>
using namespace std;

class Trie{
    vector< vector<int> > T;
    vector<bool> endOfWord;
    int prox = 1;
public:
    Trie(int maxNodes, int alphabetSize){ //maxNodes is the
       maximum amount of letters allowed
        T = vector< vector<int> >(maxNodes,
           vector<int>(alphabetSize, 0));
        endOfWord = vector<bool>(maxNodes, false);
    }

    void insert(string s){
        int at = 1;
        for(int i = 0; i<s.length(); ++i){
            int let = s[i] - 'a'; //base letter of the
               alphabet
            if(T[at][let] == 0){
                T[at][let] = ++this->prox;
            }
            at = T[at][let];
            if(i == s.length() - 1) endOfWord[at] = true;
        }
    }

    bool search(string s){ //retorna verdadeiro se s
       prefixo de algu m que est  na TRIE
        int at = 1;
        for(int i = 0; i<s.length(); ++i){
            int let = s[i] - 'a';
            if(T[at][let] == 0) return false;
            at = T[at][let];
        }
        return true;
    }
};
```

6 Geometria

6.1 Primitivas

```
typedef double ld;

const int INF = 0x3f3f3f3f;
const ll LINF = 0x3f3f3f3f3f3f3f3fll;
const ld DINF = 1e18;
const ld pi = acos(-1.0);
const ld eps = 1e-9;

bool eq(ld a, ld b) {
    return abs(a - b) <= eps;
}

struct pt { // ponto
    ld x, y;
    pt() {}
    pt(ld x, ld y) : x(x), y(y) {}
    bool operator < (const pt p) const {
        if (!eq(x, p.x)) return x < p.x;
        return y < p.y;
    }
    bool operator == (const pt p) const {
        return eq(x, p.x) and eq(y, p.y);
    }
    pt operator + (const pt p) const { return pt(x+p.x,
        y+p.y); }
    pt operator - (const pt p) const { return pt(x-p.x,
        y-p.y); }
    pt operator * (const ld c) const { return pt(x*c  , y*c
        ); }
    pt operator / (const ld c) const { return pt(x/c  , y/c
        ); }
};

struct line { // reta
    pt p, q;
    line() {}
    line(pt p, pt q) : p(p), q(q) {}
};

// PONTO & VETOR

ld dist(pt p, pt q) { // distancia
    return sqrt(sq(p.x - q.x) + sq(p.y - q.y));
}

ld dist2(pt p, pt q) { // quadrado da distancia
    return sq(p.x - q.x) + sq(p.y - q.y);
}

ld norm(pt v) { // norma do vetor
    return dist(pt(0, 0), v);
}

pt normalize(pt v) { // vetor normalizado
    if (!norm(v)) return v;
    v = v / norm(v);
    return v;
}

ld dot(pt u, pt v) { // produto escalar
    return u.x * v.x + u.y * v.y;
}

ld cross(pt u, pt v) { // norma do produto vetorial
    return u.x * v.y - u.y * v.x;
}

ld sarea(pt p, pt q, pt r) { // area com sinal
    return cross(q - p, r - q) / 2;
}

bool col(pt p, pt q, pt r) { // se p, q e r sao colin.
    return eq(sarea(p, q, r), 0);
}
```

```
int paral(pt u, pt v) { // se u e v sao paralelos
    u = normalize(u);
    v = normalize(v);
    if (eq(u.x, v.x) and eq(u.y, v.y)) return 1;
    if (eq(u.x, -v.x) and eq(u.y, -v.y)) return -1;
    return 0;
}

bool ccw(pt p, pt q, pt r) { // se p, q, r sao ccw
    return sarea(p, q, r) > 0;
}

pt rotate(pt p, ld th) { // rotaciona o ponto th radianos
    return pt(p.x * cos(th) - p.y * sin(th),
              p.x * sin(th) + p.y * cos(th));
}

pt rotate90(pt p) { // rotaciona 90 graus
    return pt(-p.y, p.x);
}

// RETA

bool isvert(line r) { // se r eh vertical
    return eq(r.p.x, r.q.x);
}

ld getm(line r) { // coef. ang. de r
    if (isvert(r)) return DINF;
    return (r.p.y - r.q.y) / (r.p.x - r.q.x);
}

ld getn(line r) { // coef. lin. de r
    if (isvert(r)) return DINF;
    return r.p.y - getm(r) * r.p.x;
}

bool lineeq(line r, line s) { // r == s
    return col(r.p, r.q, s.p) and col(r.p, r.q, s.q);
}
```

```
bool paraline(line r, line s) { // se r e s sao paralelas
    if (isvert(r) and isvert(s)) return 1;
    if (isvert(r) or isvert(s)) return 0;
    return eq(getm(r), getm(s));
}

bool isinline(pt p, line r) { // se p pertence a r
    return col(p, r.p, r.q);
}

bool isinseg(pt p, line r) { // se p pertence ao seg de r
    if (p == r.p or p == r.q) return 1;
    return paral(p - r.p, p - r.q) == -1;
}

pt proj(pt p, line r) { // projecao do ponto p na reta r
    if (r.p == r.q) return r.p;
    r.q = r.q - r.p; p = p - r.p;
    pt proj = r.q * (dot(p, r.q) / dot(r.q, r.q));
    return proj + r.p;
}

pt inter(line r, line s) { // r inter s
    if (paraline(r, s)) return pt(DINF, DINF);

    if (isvert(r)) return pt(r.p.x, getm(s) * r.p.x +
       getn(s));
    if (isvert(s)) return pt(s.p.x, getm(r) * s.p.x +
       getn(r));

    ld x = (getn(s) - getn(r)) / (getm(r) - getm(s));
    return pt(x, getm(r) * x + getn(r));
}
```

```
bool interseg(line r, line s) { // se o seg de r intercepta
    o seg de s
    if (paraline(r, s)) {
        return isinseg(r.p, s) or isinseg(r.q, s)
            or isinseg(s.p, r) or isinseg(s.q, r);
    }
    pt i = inter(r, s);
    return isinseg(i, r) and isinseg(i, s);
}

ld disttoline(pt p, line r) { // distancia do ponto a reta
    return dist(p, proj(p, r));
}

ld disttoseg(pt p, line r) { // distancia do ponto ao seg
    if (isinseg(proj(p, r), r))
        return disttoline(p, r);
    return min(dist(p, r.p), dist(p, r.q));
}

ld distseg(line a, line b) { // distancia entre seg
    if (interseg(a, b)) return 0;

    ld ret = DINF;
    ret = min(ret, disttoseg(a.p, b));
    ret = min(ret, disttoseg(a.q, b));
    ret = min(ret, disttoseg(b.p, a));
    ret = min(ret, disttoseg(b.q, a));

    return ret;
}

// POLIGONO

ld polper(vector<pt> v) { // perimetro do poligono
    ld ret = 0;
    for (int i = 0; i < sz(v); i++)
        ret += dist(v[i], v[(i + 1) % sz(v)]);
    return ret;
}

ld polarea(vector<pt> v) { // area do poligono
    ld ret = 0;
    for (int i = 0; i < sz(v); i++)
        ret += sarea(pt(0, 0), v[i], v[(i + 1) % sz(v)]);
    return abs(ret);
}

bool onpol(pt p, vector<pt> v) { // se um ponto esta na
    fronteira do poligono
    for (int i = 0; i < sz(v); i++)
        if (isinseg(p, line(v[i], v[(i + 1) % sz(v)])))
            return 1;
    return 0;
}

bool inpol(pt p, vector<pt> v) { // se um ponto pertence ao
    poligono
    if (onpol(p, v)) return 1;
    int c = 0;
    line r = line(p, pt(DINF, pi * DINF));
    for (int i = 0; i < sz(v); i++) {
        line s = line(v[i], v[(i + 1) % sz(v)]);
        if (interseg(r, s)) c++;
    }
    return c & 1;
}

bool interpol(vector<pt> v1, vector<pt> v2) { // se dois
    poligonos se interceptam
    for (int i = 0; i < sz(v1); i++) if (inpol(v1[i], v2))
        return 1;
    for (int i = 0; i < sz(v2); i++) if (inpol(v2[i], v1))
        return 1;
    return 0;
}

ld distpol(vector<pt> v1, vector<pt> v2) { // distancia
    entre poligonos
    if (interpol(v1, v2)) return 0;
```

```
    ld ret = DINF;

    for (int i = 0; i < sz(v1); i++) for (int j = 0; j <
       sz(v2); j++)
        ret = min(ret, distseg(line(v1[i], v1[(i + 1) %
           sz(v1)]),
                    line(v2[j], v2[(j + 1) % sz(v2)])));
    return ret;
}

vector<pt> convexhull(vector<pt> v) { // convex hull
    vector<pt> l, u;

    sort(v.begin(), v.end());

    for (int i = 0; i < sz(v); i++) {
        while (sz(l) > 1 and !ccw(v[i], l[sz(l) - 1],
           l[sz(l) - 2]))
            l.pop_back();
        l.pb(v[i]);
    }
    for (int i = sz(v) - 1; i >= 0; i--) {
        while (sz(u) > 1 and !ccw(v[i], u[sz(u) - 1],
           u[sz(u) - 2]))
            u.pop_back();
        u.pb(v[i]);
    }

    l.pop_back(); u.pop_back();

    for (int i = 0; i < sz(u); i++) l.pb(u[i]);

    return l;
}
```

```
// CIRCULO
pt getcenter(pt a, pt b, pt c) { // centro da circunferencia
   dado 3 pontos
    b = (a + b) / 2;
    c = (a + c) / 2;
    return inter(line(b, b + rotate90(a - b)),
            line(c, c + rotate90(a - c)));
}
circle minCirc(vector<PT> v) { // minimum enclosing circle
    int n = v.size();
    random_shuffle(v.begin(), v.end());
    PT p = PT(0.0, 0.0);
    circle ret = circle(p, 0.0);
    for(int i = 0; i < n; i++) {
        if(!inside(ret, v[i])) {
            ret = circle(v[i], 0);
            for(int j = 0; j < i; j++) {
                if(!inside(ret, v[j])) {
                    ret = circle((v[i] + v[j]) / 2.0,
                       sqrt(dist2(v[i], v[j])) / 2.0);
                    for(int k = 0; k < j; k++) {
                        if(!inside(ret, v[k])) {
                            p = bestOf3(v[i], v[j], v[k]);
                            ret = circle(p, sqrt(dist2(p,
                               v[i])));
                        }
                    }
                }
            }
        }
    }
    return ret;
}
// comparador pro set para fazer sweep angle com segmentos
double ang;
struct cmp {
    bool operator () (const line& a, const line& b) {
        line r = line(pt(0, 0), rotate(pt(1, 0), ang));
        return norm(inter(r, a)) < norm(inter(r, b));
    }
};
```

7 Diversos

7.1 Busca Ternária

```
// A fun   o deve ser estritamente crescente e depois
   estritamente decrescente (max) ou
// estritamente decrescente e depois estritamente crescente
   (min)

// Complexidade - O(log(r-l))

const double eps = 1e-9;
// Com ponto flutuante - max
double ts( double l, double r ){
    while( r-l > eps ){
        double m1 = l + ((r-l)/3);
        double m2 = r - ((r-l)/3);
        double f1 = f(m1);
        double f2 = f(m2);
        if( f1 < f2 ){  // min -> f1 > f2
            l = m1;
        }else{
            r = m2;
        }
    }
    return f(l);
}
// Com inteiros - max
int ts( int l, int r ){
  while( l < r ){
    int m = (l + r)/2;
      if( f(m) > f(m+1) ) {  // min -> f(m) < f(m+1)
        r = m;
      }else{
        l = m+1;
      }
   }
   return f(l+1); // min -> f(l)
}
```

7.2 Random Shuffle

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
using namespace std;

mt19937 rng((int)
   chrono::steady_clock::now().time_since_epoch().count());

int main(){
    ios::sync_with_stdio(false);

    uniform_int_distribution<int> dist {0,1}; // intervalo
       dos numeros gerados
    auto gen = [&dist]() { return dist(rng); };

    vector<int> X(10);
    generate(X.begin(), X.end(), gen);

    for(int x : X) cout << x << ' ';
    cout << '\n';

    return 0;
}
```

8 Extra

8.1 vimrc

```
set ts=4 si ai sw=4 number mouse=a
syntax on
```

8.2 Makefile

```
CXX = g++
CXXFLAGS = -O2 -Wall -Wshadow -std=c++11 -Wno-unused-result
   -Wno-sign-compare
```

8.3 Template

```
#include <bits/stdc++.h>

using namespace std;

#define endl '\n'

#define pb push_back
#define f first
#define s second
#define BUFF ios::sync_with_stdio(false);

typedef long long int ll;
typedef long double ld;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;

const int INF = 0x3f3f3f3f;
const ll LINF = 0x3f3f3f3f3f3f3f3fll;

mt19937 rng((int)
   chrono::steady_clock::now().time_since_epoch().count());

const int MAX = (int)1e5+10;

int main(){

  BUFF;



  return 0;
}
```