SamuellH12 - ICPC Library

Conteúdo

1 Data Structures

1.1 BIT

```
struct BIT {
  vector<int> bit;
  int N;

  BIT(){}
  BIT(int n) : N(n+1), bit(n+1){}

  void update(int pos, int val){
    for(; pos < N; pos += pos&(-pos))
      bit[pos] += val;
  }

  int query(int pos){
    int sum = 0;
    for(; pos > 0; pos -= pos&(-pos))
      sum += bit[pos];
    return sum;
  }
};
```

1.2 BIT2D

```
const int MAXN = 1e3 + 5;

struct BIT2D {
  int bit[MAXN][MAXN];

  void update(int X, int Y, int val){
    for(int x = X; x < MAXN; x += x&(-x))
      for(int y = Y; y < MAXN; y += y&(-y))
        bit[x][y] += val;
  }

  int query(int X, int Y){
    int sum = 0;
    for(int x = X; x > 0; x -= x&(-x))
      for(int y = Y; y > 0; y -= y&(-y))
        sum += bit[x][y];
    return sum;
  }

  void updateArea(int xi, int yi, int xf, int yf, int val){
    update(xi,   yi,    val);
    update(xf+1, yi,   -val);
    update(xi,   yf+1, -val);
    update(xf+1, yf+1,  val);
  }

  int queryArea(int xi, int yi, int xf, int yf){
    return query(xf, yf) - query(xf, yi-1) - query(xi-1, yf) + query(xi-1,
        yi-1);
  }
};

/* Complexity: O(Log^2 N)
Bit.update(x, y, v);  //Adiciona +v na posi  o {x, y} da BIT
Bit.query(x, y);    //Retorna o somatorio do retangulo de inicio {1, 1} e
    fim {x, y}
Bit.queryArea(xi, yi, xf, yf);     //Retorna o somatorio do retangulo de
    inicio {xi, yi} e fim {xf, yf}
Bit.updateArea(xi, yi, xf, yf, v); //adiciona +v no retangulo de inicio {xi
    , yi} e fim {xf, yf}

IMPORTANTE! UpdateArea N O atualiza o valor de todas as c lulas no
    ret ngulo!!! Deve ser usado para Color Update
IMPORTANTE! Use query(x, y) Para acessar o valor da posi  o (x, y) quando
     estiver usando UpdateArea
IMPORTANTE! Use queryArea(x, y, x, y) Para acessar o valor da posi  o (x,
     y) quando estiver usando Update Padr o  */
```

1.3 BIT2D Sparse

```
#define pii pair<int, int>
#define upper(v, x) (upper_bound(begin(v), end(v), x) - begin(v))

struct BIT2D {
  vector<int> ord;
  vector<vector<int>> bit, coord;

  BIT2D(vector<pii> pts){
    sort(begin(pts), end(pts));

    for(auto [x, y] : pts)
      if(ord.empty() || x != ord.back())
        ord.push_back(x);

    bit.resize(ord.size() + 1);
    coord.resize(ord.size() + 1);

    sort(begin(pts), end(pts), [&](pii &a, pii &b){
      return a.second < b.second;
    });

    for(auto [x, y] : pts)
      for(int i=upper(ord, x); i < bit.size(); i += i&-i)
        if(coord[i].empty() || coord[i].back() != y)
          coord[i].push_back(y);

    for(int i=0; i<bit.size(); i++) bit[i].assign(coord[i].size()+1, 0);
  }

  void update(int X, int Y, int v){
    for(int i = upper(ord, X); i<bit.size(); i += i&-i)
      for(int j = upper(coord[i], Y); j < bit[i].size(); j += j&-j)
        bit[i][j] += v;
  }

  int query(int X, int Y){
    int sum = 0;
    for(int i = upper(ord, X); i > 0; i -= i&-i)
      for(int j = upper(coord[i], Y); j > 0; j -= j&-j)
        sum += bit[i][j];
    return sum;
  }

  void updateArea(int xi, int yi, int xf, int yf, int val){
    update(xi,   yi,    val);
    update(xf+1, yi,   -val);
    update(xi,   yf+1, -val);
    update(xf+1, yf+1,  val);
  }

  int queryArea(int xi, int yi, int xf, int yf){
    return query(xf, yf) - query(xf, yi-1) - query(xi-1, yf) + query(xi-1,
        yi-1);
  }
};

/***************************************
Sparse Binary Indexed Tree 2D

Recebe o conjunto de pontos que ser o usados para fazer os updates e
as queries e cria uma BIT 2D esparsa que independe do "tamanho do grid".

Build: O(N Log N) (N -> Quantidade de Pontos)
Query/Update: O(Log N)

BIT2D(pts);   // pts -> vecotor<pii> com todos os pontos em que ser o
    feitas queries ou updates

Credits: TFG (TFG50 on Git: https://github.com/tfg50/Competitive-
    Programming/blob/master/Biblioteca/Data%20Structures/Bit2D.cpp)
***************************************/
```

1.4 Prefix Sum 2D

```
const int MAXN = 1e3 + 5;
int ps [MAXN][MAXN];

void calcPS2d(){
  for (int i = 1; i < MAXN; i++) ps[0][i] += ps[0][i - 1];  //inicializo a
      la linha
  for (int i = 1; i < MAXN; i++) ps[i][0] += ps[i - 1][0];  //inicializo a
      la coluna

  for (int i = 1; i < MAXN; i++)
    for (int j = 1; j < MAXN; j++)
      ps[i][j] += ps[i - 1][j] + ps[i][j - 1] - ps[i - 1][j - 1];
}
int queryPS2d(int xi, int yi, int xf, int yf){ return ps[xf][yf] - ps[xf][
    yi-1] - ps[xi-1][yf] + ps[xi-1][yi-1]; }

/**************
Complexidade:
-> Calcular:  O(N^2)
-> Queries:   O(1)
***************/
```

1.5 SegTree

```
const int MAXN = 1e6 + 5;
int seg[4*MAXN];

int query(int no, int l, int r, int a, int b){
  if(b <  l || r <  a) return 0;
  if(a <= l && r <= b) return seg[no];

  int m=(l+r)/2, e=no*2, d=no*2+1;

  return query(e, l, m, a, b) + query(d, m+1, r, a, b);
}

void update(int no, int l, int r, int pos, int v){
  if(pos < l || r < pos) return;
  if(l == r){seg[no] = v; return; }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  update(e, l,   m, pos, v);
  update(d, m+1, r, pos, v);

  seg[no] = seg[e] + seg[d];
}

void build(int no, int l, int r, vector<int> &lista){
  if(l == r){ seg[no] = lista[l]; return; }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  build(e, l,   m, lista);
  build(d, m+1, r, lista);

  seg[no] = seg[e] + seg[d];
}

/******************************************************
-> Segment Tree com:
  - Query em Range
```

```
  - Update em Ponto

build (1, 1, n, lista);
query (1, 1, n, a, b);
update(1, 1, n, i, x);

|   n    | tamanho
| [a, b] | intervalo da busca
|   i    | posi  o a ser modificada
|   x    | novo valor da posi  o i
| lista  | vector de elementos originais

Build:  O(N)
Query:  O(log N)
Update: O(log N)
*******************************************************/
```

1.6 SegTree Lazy

```
const int MAXN = 1e6 + 5;
int seg[4*MAXN];
int lazy[4*MAXN];

void unlazy(int no, int l, int r){
  if(lazy[no] == 0) return;

  int m=(l+r)/2, e=no*2, d=no*2+1;

  seg[no] += (r-l+1) * lazy[no];

  if(l != r){
    lazy[e] += lazy[no];
    lazy[d] += lazy[no];
  }

  lazy[no] = 0;
}

int query(int no, int l, int r, int a, int b){
  unlazy(no, l, r);
  if(b <  l || r <  a) return 0;
  if(a <= l && r <= b) return seg[no];

  int m=(l+r)/2, e=no*2, d=no*2+1;

  return query(e, l, m, a, b) + query(d, m+1, r, a, b);
}

void update(int no, int l, int r, int a, int b, int v){
  unlazy(no, l, r);
  if(b <  l || r <  a) return;
  if(a <= l && r <= b)
  {
    lazy[no]+= v;
    unlazy(no, l, r);
    return;
  }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  update(e, l,   m, a, b, v);
  update(d, m+1, r, a, b, v);

  seg[no] = seg[e] + seg[d];
}

void build(int no, int l, int r, vector<int> &lista){
  if(l == r){ seg[no] = lista[l-1]; return; }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  build(e, l,   m, lista);
  build(d, m+1, r, lista);

  seg[no] = seg[e] + seg[d];
}

/*******************************************************
-> Segment Tree - Lazy Propagation com:
  - Query em Range
  - Update em Range

build (1, 1, n, lista);
query (1, 1, n, a, b);
update(1, 1, n, a, b, x);

|   n    | o tamanho m ximo da lista
| [a, b] | o intervalo da busca ou update
|   x    | o novo valor a ser somada no intervalo [a, b]
| lista  | o array de elementos originais

Build:  O(N)
Query:  O(log N)
Update: O(log N)
Unlazy: O(1)
*******************************************************/
```

1.7 SegTree Iterativa

```
template<typename T> struct SegTree {
  int n;
  vector<T> seg;
  T join(T&l, T&r){ return l + r; }

  void init(vector<T>&base){
    n = base.size();
    seg.resize(2*n);
    for(int i=0; i<n; i++) seg[i+n] = base[i];
    for(int i=n-1; i>0; i--) seg[i] = join(seg[i*2], seg[i*2+1]);
  }

  T query(int l, int r){ //[L, R] & [0, n-1]
    T ans = 0; //NEUTRO //if order matters, change to l_ans, r_ans
    for(l+=n, r+=n+1; l<r; l/=2, r/=2){
      if(l&1) ans = join(ans, seg[l++]);
      if(r&1) ans = join(seg[--r], ans);
    }
    return ans;
  }

  void update(int i, T v){
    for(seg[i+=n] = v; i/=2;) seg[i] = join(seg[i*2], seg[i*2+1]);
  }
};
```

1.8 SegTree Lazy Iterativa

```
template<typename T> struct SegTree {
  int n, h;
  vector<T> seg, lzy;
  vector<int> sz;
  T join(T&l, T&r){ return l + r; }

  void init(int _n){
    n = _n;
    h = 32 - __builtin_clz(n);
    seg.resize(2*n);
```

```
    lzy.resize(n);
    sz.resize(2*n, 1);
    for(int i=n-1; i; i--) sz[i] = sz[i*2] + sz[i*2+1];
    // for(int i=0; i<n; i++) seg[i+n] = base[i];
    // for(int i=n-1; i; i--) seg[i] = join(seg[i*2], seg[i*2+1]);
  }

  void apply(int p, T v){
    seg[p] += v * sz[p];
    if(p < n) lzy[p] += v;
  }
  void push(int p){
    for(int s=h, i=p>>s; s; s--, i=p>>s)
      if(lzy[i] != 0) {
        apply(i*2,   lzy[i]);
        apply(i*2+1, lzy[i]);
        lzy[i] = 0; //NEUTRO
      }
  }
  void build(int p) {
    for(p/=2; p; p/= 2){
      seg[p] = join(seg[p*2], seg[p*2+1]);
      if(lzy[p] != 0) seg[p] += lzy[p] * sz[p];
    }
  }

  T query(int l, int r){ //[L, R] & [0, n-1]
    l+=n, r+=n+1;
    push(l); push(r-1);

    T ans = 0; //NEUTRO
    for(; l<r; l/=2, r/=2){
      if(l&1) ans = join(seg[l++], ans);
      if(r&1) ans = join(ans, seg[--r]);
    }
    return ans;
  }

  void update(int l, int r, T v){
    l+=n, r+=n+1;
    push(l); push(r-1);

    int l0 = l, r0 = r;
    for(; l<r; l/=2, r/=2){
      if(l&1) apply(l++, v);
      if(r&1) apply(--r, v);
    }
    build(l0); build(r0-1);
  }
};
```

1.9 SegTree Persistente

```
struct Node {
  int val = 0;
  Node *L = NULL, *R = NULL;
  Node(int v = 0) : val(v), L(NULL), R(NULL) {}
};

Node* build(int l, int r){
  if(l == r) return new Node();

  int m = (l+r)/2;

  Node *node = new Node();

  node->L = build(l,   m);
  node->R = build(m+1, r);
  node->val = node->L->val + node->R->val;

  return node;
}

Node* update(Node *node, int l, int r, int pos, int v){
  if( pos < l || r < pos ) return node;
  if(l == r) return new Node(node->val + v);

  int m = (l+r)/2;

  if(!node->L) node->L = new Node();
  if(!node->R) node->R = new Node();

  Node *nw = new Node();

  nw->L = update(node->L, l,   m, pos, v);
  nw->R = update(node->R, m+1, r, pos, v);
  nw->val = nw->L->val + nw->R->val;

  return nw;
}

int query(Node *node, int l, int r, int a, int b){
  if(b <  l || r <  a) return 0;
  if(a <= l && r <= b) return node->val;

  int m = (l+r)/2;

  if(!node->L) node->L = new Node();
  if(!node->R) node->R = new Node();

  return query(node->L, l, m, a, b) + query(node->R, m+1, r, a, b);
}

int kth(Node *Left, Node *Right, int l, int r, int k){
  if(l == r) return l;

  int sum = Right->L->val - Left->L->val;
  int m = (l+r)/2;

  if(sum >= k) return kth(Left->L, Right->L, l, m, k);

  return kth(Left->R, Right->R, m+1, r, k - sum);
}

/*******************************************************
-> Segment Tree Persistente com:
  - Query em Range
  - Update em Ponto

Build(1, N) -> Cria uma Seg Tree completa de tamanho N; RETORNA um *
    Ponteiro pra Ra z
Update(Root, 1, N, pos, v)  -> Soma +V na p o s i   o POS; RETORNA um *
    Ponteiro pra Ra z da nova vers o;
Query(Root, 1, N, a, b)   -> RETORNA o valor calculado no range [a, b];
Kth(RootL, RootR, 1, N, K)  -> Faz uma Busca Bin ria na Seg; Mais detalhes
     abaixo;

[ Root -> N   Raiz da Vers o da Seg na qual se quer realizar a o p e r a   o
    ]

Para guardar as Ra zes, use:
-> vector<Node*> roots; ou
-> Node* roots [MAXN];

Build:  O(N)
Query:  O(log N)
Update: O(log N)
Kth:  O(Log N)

Comportamento do K-th(SegL, SegR, 1, N, K):
  -> Retorna  ndice  da primeira p o s i   o i cuja soma de prefixos [1, i]
        >= k
  na Seg resultante da s u b t r a   o dos valores da (Seg R) - (Seg L).
  -> Pode ser utilizada para consultar o K- simo  menor valor no intervalo
      [L, R] de um array.
  Para isso a Seg deve ser utilizada como um array de frequ ncias. Comece
      com a Seg zerada (Build).
```

```
  Para cada valor V do Array chame um update(roots.back(), 1, N, V, 1) e
      guarde o ponteiro da seg.
  Para consultar o K- simo  menor valor de [L, R] chame kth(roots[L-1],
      roots[R], 1, N, K);

IMPORTANTE! Cuidado com o Kth ao acessar uma Seg que e s t  esparsa (RTE).
    Nesse caso,
chame o Build antes ou garanta criar os  n  s  L e R antes de acess -los (
    como na query).
********************************************************/
```

1.10 Sparse Table

```
const int MAXN = 1e5 + 5;
const int MAXLG = 31 - __builtin_clz(MAXN) + 1;

int value[MAXN], table[MAXLG][MAXN];

void build(int N){
  for(int i=0; i<N; i++) table[0][i] = value[i];

  for(int p=1; p < MAXLG; p++)
    for(int i=0; i + (1 << p) <= N; i++)
      table[p][i] = min(table[p-1][i], table[p-1][i+(1 << (p-1))]);
}

int query(int l, int r){
  int p = 31 - __builtin_clz(r - l + 1);  //floor log
  return min(table[p][l], table[p][ r - (1<<p) + 1 ]);
}
/**************************************************
Sparse Table for Range Minimum Query [L, R] [0, N)
build:  O(N log N)
Query:  O(1)
Value -> Original Array
**************************************************/
```

2 dp

2.1 Digit DP

```
#define ll long long
using namespace std;

ll dp[2][19][170];

int limite[19];
ll digitDP(int idx, int sum, bool flag){
    if(idx < 0) return sum;
    if(~dp[flag][idx][sum]) return dp[flag][idx][sum];

    dp[flag][idx][sum] = 0;
  int lm = flag ? limite[idx] : 9;

    for(int i=0; i<=lm; i++)
        dp[flag][idx][sum] += digitDP(idx-1, sum+i, (flag && i == lm));

    return dp[flag][idx][sum];
}

ll solve(ll k){
    memset(dp, -1, sizeof dp);

  int sz=0;
  while(k){
    limite[sz++] = k % 10LL;
    k /= 10LL;
  }

  return digitDP(sz-1, 0, true);
}

/*********************
Digit DP - Sum of Digits

Solve(K)  -> Retorna a soma dos d gitos de todo n mero X tal que: 0 <= X
     <= K
dp[D][S][f]  -> D: Quantidade de d gitos; S: Soma dos d gitos; f: Flag
    que indica o limite.
int limite[D] -> Guarda os d gitos de K.

Complexity: O(D^2 * B^2)  (B = Base = 10)
********************/
```

2.2 LIS

```
int LIS(vector<int>& nums){
  vector<int> lis;

  for(auto x : nums)
  {
    auto it = lower_bound(lis.begin(), lis.end(), x);

    if(it == lis.end()) lis.push_back(x);
    else *it = x;
  }

  return (int) lis.size();
}
/***************************
LIS - Longest Increasing Subsequence

Complexity: O(N Log N)
* For ICREASING sequence, use lower_bound()
* For NON DECREASING sequence, use upper_bound()
****************************/
```

2.3 SOS DP

```
#define ll long long
using namespace std;

const int N = 20;
ll dp[1<<N], iVal[1<<N];

void sosDP() // O(N * 2^N)
{
    for(int i=0; i<(1<<N); i++)
        dp[i] = iVal[i];

  for(int i=0; i<N; i++)
    for(int mask=0; mask<(1<<N); mask++)
      if(mask&(1<<i))
        dp[mask] += dp[mask^(1<<i)];
}

/********************
SOS DP - Sum over Subsets

Dado que cada mask possui um valor inicial (iVal), computa
para cada mask a soma dos valores de todas as suas submasks.

N -> N mero  M ximo de Bits
iVal[mask] -> initial Value / Valor Inicial da Mask
```

```
dp[mask]   -> Soma de todos os SubSets

Iterar por todas as submasks: for(int sub=mask; sub>0; sub=(sub-1)&mask)
********************/
```

3 Geometry

3.1 ConvexHull

```
#define ll long long
using namespace std;

struct PT {
  ll x, y;
  PT(ll x=0, ll y=0) : x(x), y(y) {}

  PT operator- (const PT&a) const{ return PT(x-a.x, y-a.y); }
  ll operator% (const PT&a) const{ return  (x*a.y - y*a.x); } //Cross // Vector product

  bool operator==(const PT&a) const{ return x == a.x && y == a.y; }
  bool operator< (const PT&a) const{ return x != a.x ? x < a.x : y < a.y; }
};

// Colinear? Mude >= 0 para > 0 nos while
vector<PT> ConvexHull(vector<PT> pts, bool sorted=false) {
  if(!sorted) sort(begin(pts), end(pts));
  pts.resize(unique(begin(pts), end(pts)) - begin(pts));
  if(pts.size() <= 1) return pts;

  int s=0, n=pts.size();
  vector<PT> h (2*n+1);

  for(int i=0; i<n; h[s++] = pts[i++])
    while(s > 1 && (pts[i] - h[s-2]) % (h[s-1] - h[s-2]) >= 0 )
      s--;

  for(int i=n-2, t=s; ~i; h[s++] = pts[i--])
    while(s > t && (pts[i] - h[s-2]) % (h[s-1] - h[s-2]) >= 0 )
      s--;

  h.resize(s-1);
  return h;
}
/********************
// FOR DOUBLE POINT //
See Geometry - General
********************/
```

3.2 Geometry - General

```
#define ll long long
#define ld long double
using namespace std;

// !!! NOT TESTED !!! //

struct PT {
  ll x, y;
  PT(ll x=0, ll y=0) : x(x), y(y) {}

  PT operator+ (const PT&a) const{ return PT(x+a.x, y+a.y); }
  PT operator- (const PT&a) const{ return PT(x-a.x, y-a.y); }
  ll operator* (const PT&a) const{ return  (x*a.x + y*a.y); } //DOT product // norm // lenght^2 // inner
  ll operator% (const PT&a) const{ return  (x*a.y - y*a.x); } //Cross // Vector product
  PT operator* (ll c) const{ return PT(x*c, y*c); }
  PT operator/ (ll c) const{ return PT(x/c, y/c); }

  bool operator==(const PT&a) const{ return x == a.x && y == a.y; }
  bool operator< (const PT&a) const{ return x != a.x ? x < a.x : y < a.y; }
  bool operator<<(const PT&a) const{ PT p=*this; return (p%a == 0) ? (p*p < a*a) : (p%a < 0);  } //angle(p) < angle(a)
};

/********************
// FOR DOUBLE POINT //
const ld EPS = 1e-9;
bool eq(ld a, ld b){ return abs(a-b) < EPS; } // ==
bool lt(ld a, ld b){ return a + EPS < b;    } // <
bool gt(ld a, ld b){ return a > b + EPS;    } // >
bool le(ld a, ld b){ return a < b + EPS;    } // <=
bool ge(ld a, ld b){ return a + EPS > b;    } // >=
bool operator==(const PT&a) const{ return eq(x, a.x) && eq(y, a.y); } // for double point
bool operator< (const PT&a) const{ return eq(x, a.x) ? lt(y, a.y) : lt(x, a.x); } // for double point
bool operator<<(PT&a){ PT&p=*this; return eq(p%a, 0) ? lt(p*p, a*a) : lt(p%a, 0);  } //angle(this) < angle(a)
//Change LL to LD and uncomment this
//Also, consider replacing comparisons with these functions
********************/

ld dist  (PT a, PT b){ return sqrtl((a-b)*(a-b)); }                    // distance from A to B
ld angle (PT a, PT b){ return acos((a*b) / sqrtl(a*a) / sqrtl(b*b)); } // Angle between A and B
PT rotate(PT p, double ang){ return PT(p.x*cos(ang) - p.y*sin(ang), p.x*sin(ang) + p.y*cos(ang)); } //Left rotation. Angle in radian

ll Area(vector<PT>& p){
  ll area = 0;
  for(int i=2; i < p.size(); i++)
    area += (p[i]-p[0]) % (p[i-1]-p[0]);
  return abs(area) / 2LL;
}

PT intersect(PT a1, PT d1, PT a2, PT d2){
  return a1 + d1 * (((a2 - a1)%d2) / (d1%d2));
}

ld dist_pt_line(PT a, PT l1, PT l2){
  return abs( ((a-l1) % (l2-l1)) / dist(l1, l2)  );
}

ld dist_pt_segm(PT a, PT s1, PT s2){
  if(s1 == s2) return dist(s1, a);

  PT d = s2 - s1;
  ld t = max(0.0L, min(1.0L, ((a-s1)*d) / sqrtl(d*d)) );

  return dist(a, s1+(d*t));
}
```

3.3 LineContainer

```
#define ll long long
using namespace std;

struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  static const ll inf = LLONG_MAX; // Double: inf = 1/.0, div(a,b) = a/b
  ll div(ll a, ll b) { return a / b - ((a ^ b) < 0 && a % b); } //floored division
```

```
  bool isect(iterator x, iterator y) {
    if(y == end()) return x->p = inf, 0;
    if(x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }

  void add_line(ll k, ll m){ // kx + m  //if minimum k*=-1, m*=-1, query*-1
    auto z = insert({k, m, 0}), y = z++, x = y;
    while(isect(y, z)) z = erase(z);
    if(x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while((y = x) != begin() && (--x)->p >= y->p) isect(x, erase(y));
  }

  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
/* Credits: kactl (https://github.com/kth-competitive-programming) */
```

4 Grafos

4.1 2SAT

```
struct TwoSat {
  int N;
  vector<vector<int>> E;

  TwoSat(int N) : N(N), E(2 * N) {}
  inline int eval(int u) const{ return u < 0 ? ((~u)+N)%(2*N) : u; }

  void add_or(int u, int v){
    E[eval(~u)].push_back(eval(v));
    E[eval(~v)].push_back(eval(u));
   }
  void add_nand(int u, int v) {
    E[eval(u)].push_back(eval(~v));
    E[eval(v)].push_back(eval(~u));
  }
  void set_true (int u){ E[eval(~u)].push_back(eval(u)); }
  void set_false(int u){ set_true(~u); }
  void add_imply(int u, int v){ E[eval(u)].push_back(eval(v)); }
  void add_and  (int u, int v){ set_true(u);  set_true(v);    }
  void add_nor  (int u, int v){ add_and(~u, ~v); }
  void add_xor  (int u, int v){ add_or(u, v); add_nand(u, v); }
  void add_xnor (int u, int v){ add_xor(u, ~v); }

  vector<bool> solve() {
    vector<bool> ans(N);
    auto scc = tarjan();

    for (int u = 0; u < N; u++)
      if(scc[u] == scc[u+N]) return {}; //false
      else ans[u] = scc[u+N] > scc[u];

    return ans; //true
  }
private:
  vector<int> tarjan() {
    vector<int> low(2*N), pre(2*N, -1), scc(2*N, -1);
    stack<int> st;
    int clk = 0, ncomps = 0;

    auto dfs = [&](auto&& dfs, int u) -> void {
      pre[u] = low[u] = clk++;
      st.push(u);

      for(auto v : E[u])
        if(pre[v] == -1) dfs(dfs, v), low[u] = min(low[u], low[v]);
        else
        if(scc[v] == -1) low[u] = min(low[u], pre[v]);

      if(low[u] == pre[u]){
        int v = -1;
        while(v != u) scc[v = st.top()] = ncomps, st.pop();
        ncomps++;
      }
    };

    for(int u=0; u < 2*N; u++)
      if(pre[u] == -1)
        dfs(dfs, u);

    return scc; //tarjan SCCs order is the reverse of topoSort, so (u->v if
         scc[v] <= scc[u])
  }
};

/***********************************
  2 SAT - Two Satisfiability Problem

IMPORTANTE! o grafo deve estar 0-indexado!

inverso de u = ~u

Retorna uma valora  o verdadeira se poss vel
Ou um vetor vazio se imposs vel;
***********************************/
```

4.2 BlockCutTree

```
#define pii pair<int,int>

const int MAXN = 1e6 + 5;
const int MAXM = 1e6 + 5;//Cuidado

vector<pii> grafo [MAXN];
int pre[MAXN], low[MAXN], clk=0, C=0;

vector<pii> edge;
bool visEdge[MAXM];
int edgeComponent[MAXM];
int vertexComponent[MAXN];

int cut[MAXN];
stack<int> s;

vector<int> tree [2*MAXN];
int componentSize[2*MAXN]; //vertex - cutPoints

void reset(int n){
  for(int i=0; i<=edge.size(); i++)
    visEdge[i] = edgeComponent[i] = 0;

  edge.clear();

  for(int i=0; i<=n; i++){
    pre[i] = low[i] = -1;
    cut[i] = false;
    vertexComponent[i] = 0;
    grafo[i].clear();
  }

  for(int i=0; i<=C; i++){
    componentSize[i] = 0;
    tree[i].clear();
  }
```

```cpp
  while(!s.empty()) s.pop();

  clk = C = 0;
}

void newComponent(int i){
  C++;
  int j;

  do {
    j = s.top(); s.pop();
    edgeComponent[j] = C;

    auto [u, v]  = edge[j];
    if(!cut[u] && !vertexComponent[u]) componentSize[C]++, vertexComponent[
        u] = C;
    if(!cut[v] && !vertexComponent[v]) componentSize[C]++, vertexComponent[
        v] = C;

  } while(!s.empty() && j != i);
}

void tarjan(int u, bool root = true){
  pre[u] = low[u] = clk++;

  bool any = false;
  int chd = 0;

  for(auto [v, i] : grafo[u]){
    if(visEdge[i]) continue;
    visEdge[i] = true;

    s.emplace(i);

    if(pre[v] == -1)
    {
      tarjan(v, false);

      low[u] = min(low[v], low[u]);
      chd++;

      if(!root && low[v] >= pre[u]) cut[u] = true, newComponent(i);
      if( root && chd >= 2)       cut[u] = true, newComponent(i);
    }
    else
      low[u] = min(low[u], pre[v]);
  }

  if(root) newComponent(-1);
}

//ATENCAO: ESTA 1-INDEXADO
void buildBCC(int n){
  vector<bool> marc(C+1, false);

  for(int u=1; u<=n; u++)
  {
    if(!cut[u]) continue;

    C++;
    cut[u] = C;

    for(auto [v, i] : grafo[u])
    {
      int ec = edgeComponent[i];
      if(!marc[ec])
      {
        marc[ec] = true;
        tree[cut[u]].emplace_back(ec);
        tree[ec].emplace_back(cut[u]);
      }
    }

    for(auto [v, i] : grafo[u])
      marc[edgeComponent[i]] = false;
  }
}

void addEdge(int u, int v){
  int i = edge.size();
  grafo[u].emplace_back(v, i);
  grafo[v].emplace_back(u, i);
  edge.emplace_back(u, v);
}

/***********************
Block Cut Tree - BiConnected Component

reset(n);
addEdge(u, v);
tarjan(Root);
buildBCC(n);

No fim o grafo da Block Cut Tree estara em _vector<int> tree []_
***********************/
```

4.3 Centroid Decomposition

```cpp
const int MAXN = 1e6 + 5;

vector<int> grafo[MAXN];
deque<int> distToAncestor[MAXN];

bool rem[MAXN];
int szt[MAXN], parent[MAXN];

void getDist(int u, int p, int d=0){
  for(auto v : grafo[u])
    if(v != p && !rem[v])
      getDist(v, u, d+1);
  distToAncestor[u].emplace_front(d);
}

int getSz(int u, int p){
  szt[u] = 1;
  for(auto v : grafo[u])
    if(v != p && !rem[v])
      szt[u] += getSz(v, u);
  return szt[u];
}

void dfsc(int u=0, int p=-1, int f=-1, int sz=-1){
  if(sz < 0) sz = getSz(u, -1); //starting new tree

  for(auto v : grafo[u])
    if(v != p && !rem[v] && szt[v]*2 >= sz)
      return dfsc(v, u, f, sz);

  rem[u] = true, parent[u] = f;
  getDist(u, -1, 0); //get subtree dists to centroid

  for(auto v : grafo[u])
    if(!rem[v])
      dfsc(v, u, u, -1);
}

/*****************************
Centroid Decomposition

dfsc() -> para criar a centroid tree

rem[u]    -> True se U j   foi removido (pra dfsc)
szt[u]    -> Size da sub rvore de U (pra dfsc)
parent[u] -> Pai de U na centroid tree *parent[ROOT] = -1
distToAncestor[u][i] -> Dist ncia na  rvore  original de u para
  seu i- simo  pai na centroid tree *distToAncestor[u][0] = 0

dfsc(u=node, p=parent(subtree), f=parent(centroid tree), sz=size of tree)
*****************************/
```

4.4 Dijkstra

```
const int MAXN = 1e6 + 5;
#define INF 0x3f3f3f3f
#define vi vector<int>
#define pii pair<int,int>

vector<pii> grafo [MAXN];

vi dijkstra(int s){
  vi dist (MAXN, INF); // !!! Change MAXN to N

  priority_queue<pii, vector<pii>, greater<pii>> fila;
  fila.push({0, s});
  dist[s] = 0;

  while(!fila.empty())
  {
    auto [d, u] = fila.top();
    fila.pop();

    if(d > dist[u]) continue;

    for(auto [v, c] : grafo[u])
      if( dist[v] > dist[u] + c )
      {
        dist[v] = dist[u] + c;
        fila.push({dist[v], v});
      }
  }

  return dist;
}
/***************************************
Dijkstra - Shortest Paths from Source

caminho minimo de um vertice u para todos os
outros vertices de um grafo ponderado

Complexity: O(N Log N)

dijkstra(s)       ->  s : Source, Origem. As distancias serao calculadas
    com base no vertice s
grafo[u] = {v, c};    ->  u : Vertice inicial, v : Vertice final, c : Custo
    da aresta
priority_queue<pii, vector<pii>, greater<pii>> ->  Ordena pelo menor custo
    -> {d, v} -> d : Distancia, v : Vertice
***************************************/
```

4.5 Dinic

```
#define ll long long

struct Aresta {
  int u, v; ll cap;
  Aresta(int u, int v, ll cap) : u(u), v(v), cap(cap) {}
};

struct Dinic {

  int n, source, sink;
  vector<vector<int>> adj;
  vector<Aresta> arestas;
  vector<int> level, ptr; //pointer para a pr xima aresta n o saturada de
      cada v rtice

  Dinic(int n, int source, int sink) : n(n), source(source), sink(sink) {
      adj.resize(n); }

  void addAresta(int u, int v, ll cap)
  {
    adj[u].push_back(arestas.size());
    arestas.emplace_back(u, v, cap);

    adj[v].push_back(arestas.size());
    arestas.emplace_back(v, u, 0);
  }

  ll dfs(int u, ll flow = 1e9){
    if(flow == 0) return 0;
    if(u == sink) return flow;

    for(int &i = ptr[u];  i < adj[u].size();  i++)
    {
      int atual = adj[u][i];
      int v = arestas[atual].v;

      if(level[u] + 1 != level[v]) continue;

      if(ll got = dfs(v, min(flow, arestas[atual].cap)) )
      {
        arestas[atual].cap -= got;
        arestas[atual^1].cap += got;
        return got;
      }
    }

    return 0;
  }

  bool bfs(){
    level = vector<int> (n, n);
    level[source] = 0;

    queue<int> fila;
    fila.push(source);

    while(!fila.empty())
    {
      int u = fila.front();
      fila.pop();

      for(auto i : adj[u]){
        int v = arestas[i].v;

        if(arestas[i].cap == 0 || level[v] <= level[u] + 1 ) continue;

        level[v] = level[u] + 1;
        fila.push(v);
      }
    }

    return level[sink] < n;
  }

  bool inCut(int u){ return level[u] < n; }

  ll maxFlow(){
    ll ans = 0;

    while( bfs() ){
      ptr = vector<int> (n+1, 0);

      while(ll got = dfs(source)) ans += got;
    }

    return ans;
  }
};
/***************************************
    Dinic - Max Flow Min Cut
Algoritmo de Dinitz para encontrar o Fluxo M ximo
IMPORTANTE! O algoritmo est  0-indexado

Complexity:
```

```
O( V^2 * E )      ->  caso geral
O( sqrt(V) * E )  ->  grafos com cap = 1 para toda aresta // matching
    bipartido

* Informa es:
  Crie o Dinic:
    Dinic dinic(n, source, sink);
  Adicione as Arestas:
    dinic.addAresta(u, v, capacity);
  Para calcular o Fluxo M ximo:
    dinic.maxFlow()
  Para saber se um v rtice U est  no Corte M nimo:
    dinic.inCut(u)

* Sobre o C digo:
  vector<Aresta> arestas;  -> Guarda todas as arestas do grafo e do grafo
      residual
  vector<vector<int>> adj; -> Guarda em adj[u] os  ndices  de todas as
      arestas saindo de u
  vector<int> ptr;   ->  Pointer para a pr xima aresta ainda n o visitada
       de cada v rtice
  vector<int> level; -> Dist ncia em v rtices a partir do Source. Se
      igual a N o v rtice  n o foi visitado.
  A BFS retorna se Sink    alcan avel de Source. Se n o    porque foi
      atingido o Fluxo M ximo
  A DFS retorna um poss vel aumento do Fluxo
**************************************/
/*************************************
* Use Cases of Flow

+ Minimum cut: the minimum cut is equal to maximum flow.
  i.e. to split the graph in two parts, one on the source side and another
      on sink side.
  The capacity of each edge is it weight.

+ Edge-disjoint paths: maximum number of edge-disjoint paths equals maximum
     flow of the
  graph, assuming that the capacity of each edge is one. (paths can be
      found greedily)

+ Node-disjoint paths: can be reduced to maximum flow. each node should
    appear in at most one
  path, so limit the flow through a node dividing each node in two. One
      with incoming edges,
  other with outgoing edges and a new edge from the first to the second
      with capacity 1.

+ Maximum matching (bipartite): maximum matching is equal to maximum flow.
    Add a source and
  a sink, edges from the source to every node at one partition and from
      each node of the
  other partition to the sink.

+ Minimum node cover (bipartite): minimum set of nodes such each edge has
    at least one
  endpoint. The size of minimum node cover is equal to maximum matching (
      Konigs theorem).

+ Maximum independent set (bipartite): largest set of nodes such that no
    two nodes are
  connected with an edge. Contain the nodes that aren't in "Min node cover"
       (N - MAXFLOW).

+ Minimum path cover (DAG): set of paths such that each node belongs to at
    least one path.
  - Node-disjoint: construc a matching where each node is represented by
      two nodes, a left and
    a right at the matching and add the edges (from l to r). Each edge in
        the matching
    corresponds to an edge in the path cover. The number of paths in the
        cover is (N - MAXFLOW).
  - General: almost like a minimum node-disjoint. Just add edges to the
      matching whenever there
    is an path from U to V in the graph (possibly through several edges).
  - Antichain: a set of nodes such that there is no path from any node to
      another. In a DAG, the
    size of min general path cover equals the size of maximum antichain (
        Dilworths theorem).
**************************************/
```

4.6 DSU Persistente

```cpp
const int MAXN = 1e6 + 5;
int pai[MAXN], sz[MAXN], tim[MAXN], t=1;

int find(int u, int q = INT_MAX){
  if( pai[u] == u || q < tim[u] ) return u;
  return find(pai[u], q);
}

void join(int u, int v){
  u = find(u);
  v = find(v);

  if(u == v) return;
  if(sz[v] > sz[u]) swap(u, v);

  pai[v] = u;
  tim[v] = t++;
  sz[u] += sz[v];
}

void resetDSU(){
  for(int i=0; i<MAXN; i++) sz[i] = 1, pai[i] = i;
  memset(tim, 0, sizeof tim);
}
/*************************************************
  SemiPersistent Disjoint Set Union

-> Complexity: O( Log N )
find(u, q) -> Retorna o representante do conjunto de U no tempo Q

 * N o    poss vel utilizar Path Compression
 * tim -> tempo em que o pai de U foi alterado
*************************************************/
```

4.7 DSU

```cpp
const int MAXN = 1e6 + 5;
int pai[MAXN], sz[MAXN];

int find(int u){
  return ( pai[u] == u  ?  u  :  pai[u] = find(pai[u]) );
}

void join(int u, int v){
  u = find(u);
  v = find(v);

  if(u == v) return;
  if(sz[v] > sz[u]) swap(u, v);

  pai[v] = u;
  sz[u] += sz[v];
}

void resetDSU(){
  for(int i=0; i<MAXN; i++)
    sz[i] = 1, pai[i] = i;
}
/*************************************************
  Disjoint Set Union - Union Find
-> Complexity:
 - Find: O(   (n) ) -> Inverse Ackermann function
 - Join: O(   (n) ) -> Inverse Ackermann function
   (n) <= 4 Para todos os casos pr ticos
*************************************************/
```

4.8 Euler Path

```
#define pii pair<int, int>
#define vi vector<int>

const int MAXN = 1e6 + 5;
const bool BIDIRECIONAL = true;

vector<pii> grafo[MAXN];
vector<bool> used;

void addEdge(int u, int v){
  grafo[u].emplace_back(v, used.size()); if(BIDIRECIONAL && u != v)
  grafo[v].emplace_back(u, used.size());
  used.emplace_back(false);
}

pair<vi, vi> EulerPath(int n, int src=0){
  int s=-1, t=-1;
  vector<int> selfLoop(n*BIDIRECIONAL, 0);

  if(BIDIRECIONAL)
  {
    for(int u=0; u<n; u++) for(auto&[v, id] : grafo[u]) if(u==v) selfLoop[u]++;
    for(int u=0; u<n; u++)
      if((grafo[u].size() - selfLoop[u])%2)
        if(t != -1) return {vi(), vi()};   // mais que 2 com grau  mpar
        else t = s, s = u;

    if(t == -1 && t != s) return {vi(), vi()}; // s   1 com grau  mpar
    if(s == -1 || t == src) s = src;           // se possivel, seta start como src
  }
  else
  {
    vector<int> in(n, 0), out(n, 0);

    for(int u=0; u<n; u++)
      for(auto [v, edg] : grafo[u])
        in[v]++, out[u]++;

    for(int u=0; u<n; u++)
      if(in[u] - out[u] == -1 && s == -1) s = u;  else
      if(in[u] - out[u] ==  1 && t == -1) t = u;  else
      if(in[u] !=out[u]) return {vi(), vi()};

    if(s == -1 && t == -1) s = t = src;         // se possivel, seta s como src
    if(s == -1 && t != -1) return {vi(), vi()}; // Existe S mas n o T
    if(s != -1 && t == -1) return {vi(), vi()}; // Existe T mas n o S
  }

  for(int i=0; grafo[s].empty() && i<n; i++) s =(s+1)%n; //evita s ser v rtice isolado

  /////// DFS ///////
  vector<int> path, pathId, idx(n, 0);
  stack<pii> st; // {Vertex, EdgeId}
  st.push({s, -1});

  while(!st.empty())
  {
    auto [u, edg] = st.top();
    while(idx[u] < grafo[u].size() && used[grafo[u][idx[u]].second]) idx[u]++;

    if(idx[u] < grafo[u].size())
    {
      auto [v, id] = grafo[u][idx[u]];
      used[id] = true;
      st.push({v, id});
      continue;
    }

    path.push_back(u);
    pathId.push_back(edg);
    st.pop();
  }

  pathId.pop_back();
  reverse(begin(path),   end(path));
  reverse(begin(pathId), end(pathId));

  ///  Grafo conexo ? ///
  int edgesTotal = 0;
  for(int u=0; u<n; u++) edgesTotal += grafo[u].size() + (BIDIRECIONAL ? selfLoop[u] : 0);
  if(BIDIRECIONAL) edgesTotal /= 2;
  if(pathId.size() != edgesTotal) return {vi(), vi()};
  //////////////////////

  return {path, pathId};
}
/**************************************
Euler Path - Algoritmo de Hierholzer para caminho Euleriano

Complexity: O(V + E)

IMPORTANTE! O algoritmo est  0-indexado

* Informa  es
  addEdge(u, v) -> Adiciona uma aresta de U para V
  EulerPath(n) -> Retorna o Euler Path, ou um vetor vazio se imposs vel
  vi path -> v rtices do Euler Path na ordem
  vi pathId -> id das Arestas do Euler Path na ordem

Euler em Undirected graph:
  - Cada v rtice tem um n mero par de arestas (circuito); OU
  - Exatamente dois v rtices t m um n mero  mpar  de arestas (caminho);
Euler em Directed graph:
  - Cada v rtice tem quantidade de arestas |entrada| == |sa da| (circuito); OU
  - Exatamente 1 tem |entrada|+1 == |sa da| && exatamente 1 tem |entrada| == |sa da|+1 (caminho);
* Circuito -> U    o primeiro e  ltimo
* Caminho -> U    o primeiro e V o  ltimo
**************************************/
```

4.9 HLD

```
#define ll long long
using namespace std;

const bool EDGE = false;
struct HLD {
public:
  vector<vector<int>> g; //grafo
  vector<int> sz, parent, tin, nxt;
  HLD(){}
  HLD(int n){ init(n); }
  void init(int n){
    t = 0;
    g.resize(n); tin.resize(n);
    sz.resize(n);nxt.resize(n);
    parent.resize(n);
  }
  void addEdge(int u, int v){
    g[u].emplace_back(v);
    g[v].emplace_back(u);
  }
  void build(int root=0){
    nxt[root]=root;
```

```
    dfs(root, root);
    hld(root, root);
  }

  ll query_path(int u, int v){
    if(tin[u] < tin[v]) swap(u, v);
    if(nxt[u] == nxt[v]) return qry(tin[v]+EDGE, tin[u]);
    return qry(tin[nxt[u]], tin[u]) + query_path(parent[nxt[u]], v);
  }

  void update_path(int u, int v, ll x){
    if(tin[u] < tin[v]) swap(u, v);
    if(nxt[u] == nxt[v]) return updt(tin[v]+EDGE, tin[u], x);
    updt(tin[nxt[u]], tin[u], x); update_path(parent[nxt[u]], v, x);
  }

private:
  ll qry(int l, int r){ if(EDGE && l>r) return 0;/*NEUTRO*/ } //call Seg,
      BIT, etc
  void updt(int l, int r, ll x){ if(EDGE && l>r) return; }    //call Seg,
      BIT, etc

  void dfs(int u, int p){
    sz[u] = 1, parent[u] = p;
    for(auto &v : g[u]) if(v != p) {
      dfs(v, u); sz[u] += sz[v];

      if(sz[v] > sz[g[u][0]] || g[u][0] == p)
        swap(v, g[u][0]);
    }
  }

  int t=0;
  void hld(int u, int p){
    tin[u] = t++;
    for(auto &v : g[u]) if(v != p)
      nxt[v] = (v == g[u][0] ? nxt[u] : v),
      hld(v, u);
  }

  /// OPTIONAL ///
  int lca(int u, int v){
    while(!inSubtree(nxt[u], v)) u = parent[nxt[u]];
    while(!inSubtree(nxt[v], u)) v = parent[nxt[v]];
    return tin[u] < tin[v] ? u : v;
  }
  bool inSubtree(int u, int v){ return tin[u] <= tin[v] && tin[v] < tin[u]
      + sz[u]; }
  //query/update_subtree[tin[u]+EDGE, tin[u]+sz[u]-1];
};

/**************************************
Heavy-Light Decomposition

Complexity: #Query_path: O(LogN*qry) #Update_path: O(LogN*updt)
Nodes: 0 <= u, v < N

Change qry(l, r) and updt(l, r) to call a query and update
structure of your will

HLD hld(n); //call init
hld.add_edges(u, v); //add all edges
hld.build(); //Build everthing for HLD

tin[u] -> Pos in the structure (Seg, Bit, ...)
nxt[u] -> Head/Endpoint
*************************************/
```

4.10 Kruskal

```
/*Create a DSU*/
void join(int u, int v); int find(int u);

const int MAXN = 1e6 + 5;
struct Aresta{ int u, v, c; };
bool compAresta(Aresta a, Aresta b){ return a.c < b.c; }

vector<Aresta> arestas;       //Lista de Arestas

int kruskal(){
  sort(begin(arestas), end(arestas), compAresta); //Ordena pelo custo
  int resp = 0;          //Custo total da MST

  for(auto a : arestas)
    if( find(a.u) != find(a.v) )
    {
      resp += a.c;
      join(a.u, a.v);
    }
  return resp;
}
/**************************************************************
  Kruskal - Minimum Spanning Tree
Algoritmo para encontrar a Arvore Geradora Minima (MST)
-> Complexity: O(E log E)
E : Numero de Arestas
**************************************************************/
```

4.11 LCA

```
const int MAXN = 1e4 + 5;
const int MAXLG = 16;

vector<int> grafo[MAXN];

int bl[MAXLG][MAXN], lvl[MAXN];

void dfs(int u, int p, int l=0){
  lvl[u] = l;
  bl[0][u] = p;

  for(auto v : grafo[u])
    if(v != p)
      dfs(v, u, l+1);
}

void buildBL(int N){
  for(int i=1; i<MAXLG; i++)
    for(int u=0; u<N; u++)
      bl[i][u] = bl[i-1][bl[i-1][u]];
}

int lca(int u, int v){
  if(lvl[u] < lvl[v]) swap(u, v);

  for(int i=MAXLG-1; i>=0; i--)
    if(lvl[u] - (1<<i) >= lvl[v])
      u = bl[i][u];

  if(u == v) return u;

  for(int i=MAXLG-1; i>=0; i--)
    if(bl[i][u] != bl[i][v])
      u = bl[i][u],
      v = bl[i][v];

  return bl[0][u];
}
/*************************************
  LCA - Lowest Common Ancestor - Binary Lifting
Algoritmo para encontrar o menor ancestral comum
entre dois v rtices em uma  rvore  enraizada
```

```
IMPORTANTE! O algoritmo est  0-indexado

Complexity:
buildBL() ->  O(N Log N)
lca()     ->  O(Log N)

* Informa  es
  -> Monte o grafo na lista de adjac ncias
  -> chame dfs(root, root) para calcular o pai e a altura de cada v rtice
  -> chame buildBL() para criar a matriz do Binary Lifting
  -> chame lca(u, v) para encontrar o menor ancestral comum
  bl[i][u] -> Binary Lifting com o (2^i)- simo  pai de u
  lvl[u]   -> Altura ou level de U na  rvore

* Em LCA o primeiro FOR iguala a altura de U e V
* E o segundo anda at   o primeiro v rtice de U que n o    ancestral de V
* A resposta    o pai desse v rtice
**************************************/
```

4.12 MinCostMaxFlow - MCMF

```
#define ll long long

struct Aresta {
  int u, v; ll cap, cost;
  Aresta(int u, int v, ll cap, ll cost) : u(u), v(v), cap(cap), cost(cost)
      {}
};

struct MCMF {
  const ll INF = numeric_limits<ll>::max();
  int n, source, sink;
  vector<vector<int>> adj;
  vector<Aresta> edges;
  vector<ll> dist, pot;
  vector<int> from;

  MCMF(int n, int source, int sink) : n(n), source(source), sink(sink) {
      adj.resize(n); pot.resize(n); }

  void addAresta(int u, int v, ll cap, ll cost){
    adj[u].push_back(edges.size());
    edges.emplace_back(u, v, cap, cost);

    adj[v].push_back(edges.size());
    edges.emplace_back(v, u, 0, -cost);
  }

  queue<int> q;
  vector<bool> vis;
  bool SPFA(){
    dist.assign(n, INF);
    from.assign(n, -1);
    vis.assign(n, false);

    q.push(source);
    dist[source] = 0;

    while(!q.empty()){
      int u = q.front();
      q.pop();

      vis[u] = false;

      for(auto i : adj[u]){
        if(edges[i].cap == 0) continue;
        int v = edges[i].v;
        ll cost = edges[i].cost;

        if(dist[v] > dist[u] + cost + pot[u] - pot[v]){
          dist[v] = dist[u] + cost + pot[u] - pot[v];
          from[v] = i;
          if(!vis[v]) q.push(v), vis[v] = true;
        }
      }
    }

    for(int u=0; u<n; u++)  //fix pot
      if(dist[u] < INF)
        pot[u] += dist[u];

    return dist[sink] < INF;
  }

  pair<ll, ll> augment(){
    ll flow = edges[from[sink]].cap, cost = 0; //fixed flow: flow = min(
        flow, remainder)

    for(int v=sink; v != source; v = edges[from[v]].u)
      flow = min(flow, edges[from[v]].cap),
      cost += edges[from[v]].cost;

    for(int v=sink; v != source; v = edges[from[v]].u)
      edges[from[v]].cap   -= flow,
      edges[from[v]^1].cap += flow;

    return {flow, cost};
  }

  bool inCut(int u){ return dist[u] < INF; }

  pair<ll, ll> maxFlow(){
    ll flow = 0, cost = 0;

    while( SPFA() ){
      auto [f, c] = augment();
      flow += f;
      cost += f*c;
    }
    return {flow, cost};
  }
};
```

4.13 SCC - Kosaraju

```
#define vi vector<int>
using namespace std;

const int MAXN = 1e6 + 5;

vi grafo[MAXN];
vi greve[MAXN];
vi dag[MAXN];
vi comp, order;
vector<bool> vis;
int C;

void dfs(int u){
  vis[u] = true;
  for(auto v : grafo[u])
    if(!vis[v])
      dfs(v);
  order.push_back(u);
}

void dfs2(int u){
  comp[u] = C;
  for(auto v : greve[u])
    if(comp[v] == -1)
      dfs2(v);
}

void kosaraju(int n){
  order.clear();
  comp.assign(n, -1);
```

```
  vis.assign(n, false);

  for(int v=0; v<n; v++)
    if(!vis[v])
      dfs(v);

  C = 0;
  reverse(begin(order), end(order));

  for(auto v : order)
    if(comp[v] == -1)
      dfs2(v), C++;

  //// Montar DAG  ////
  vector<bool> marc(C, false);

  for(int u=0; u<n; u++){
    for(auto v : grafo[u])
    {
      if(comp[v] == comp[u] || marc[comp[v]]) continue;

      marc[comp[v]] = true;
      dag[comp[u]].emplace_back(comp[v]);
    }

    for(auto v : grafo[u]) marc[comp[v]] = false;
  }
}
/***************************************
Kosaraju - Strongly Connected Component
Algoritmo de Kosaraju para encontrar Componentes Fortemente Conexas

Complexity: O(V + E)
IMPORTANTE! O algoritmo est  0-indexado

*** Vari veis e explica  es ***
int C   -> C    a quantidade de Componetes Conexas. As componetes est o
    numeradas de 0 a C-1
dag     -> Ap s rodar o Kosaraju, o grafo das componentes conexas s er
    criado aqui
comp[u] -> Diz a qual componente conexa U faz parte
order   -> Ordem de sa da dos v rtices. Necess rio para o Kosaraju
grafo   -> grafo direcionado
greve -> grafo reverso (que deve ser construido junto ao grafo normal) !!!

NOTA: A ordem que o Kosaraju descobre as componentes    uma Ordena  o
    Topol gica do SCC
em que o dag[0] n o possui grau de entrada e o dag[C-1] n o possui grau
    de saida
***************************************/
```

4.14 Tarjan

```
const int MAXN = 1e6 + 5;
int pre[MAXN], low[MAXN], clk=0;
vector<int> grafo [MAXN];

vector<pair<int, int>> pontes;
vector<int> cut;

// lembrar do memset(pre, -1, sizeof pre);
void tarjan(int u, int p = -1){
  pre[u] = low[u] = clk++;

  bool any = false;
  int chd = 0;

  for(auto v : grafo[u]){
    if(v == p) continue;

    if(pre[v] == -1)
    {
      tarjan(v, u);

      low[u] = min(low[v], low[u]);

      if(low[v] >  pre[u]) pontes.emplace_back(u, v);
      if(low[v] >= pre[u]) any = true;

      chd++;
    }
    else
      low[u] = min(low[u], pre[v]);
  }

  if(p == -1 && chd >= 2) cut.push_back(u);
  if(p != -1 && any)      cut.push_back(u);
}

/***************************************
  Tarjan - Pontes e Pontos de Articula  o
Algoritmo para encontrar pontes e pontos de articula  o.

Complexity: O(V + E)
IMPORTANTE! Lembre do memset(pre, -1, sizeof pre);

*** Vari veis e explica  es ***
pre[u] = "Altura", ou, x- simo  elemento visitado na DFS. Usado para saber
    a posi  o de um v rtice na  rvore  de DFS
low[u] = Low Link de U, ou a menor aresta de retorno (mais pr xima da raiz
    ) que U alcan a entre seus filhos

chd = Children. Quantidade de componentes filhos de U. Usado para saber se
    a Raiz    Ponto de Articula  o.
any = Marca se alguma aresta de retorno em qualquer dos componentes filhos
    de U n o ultrapassa U. Se isso for verdade, U    Ponto de
    Articula  o.

if(low[v] >  pre[u]) pontes.emplace_back(u, v); -> se a mais alta aresta de
     retorno de V (ou o menor low) estiver abaixo de U, ent o U-V
    ponte
if(low[v] >= pre[u]) any = true;        -> se a mais alta aresta de retorno
     de V (ou o menor low) estiver abaixo de U ou igual a U, ent o U
    Ponto de Articula  o
***************************************/
```

5 Math

5.1 fexp

```
#define ll long long
ll MOD = 1e9 + 7;

ll fexp(ll b, ll p){
  ll ans = 1;

  while(p){
    if(p&1) ans = (ans*b) % MOD;
    b = b * b % MOD;
    p >>= 1;
  }

  return ans % MOD;
}
// O(Log P) // b - Base // p - Pot ncia
```

6 others

6.1 Hungarian

```
typedef int TP;

const int MAXN = 1e3 + 5;
const TP INF = 0x3f3f3f3f;

TP matrix[MAXN][MAXN];
TP row[MAXN], col[MAXN];
int match[MAXN], way[MAXN];

TP hungarian(int n, int m){
  memset(row, 0, sizeof row);
  memset(col, 0, sizeof col);
  memset(match, 0, sizeof match);

  for(int i=1; i<=n; i++)
  {
    match[0] = i;
    int j0 = 0, j1, i0;
    TP delta;

    vector<TP> minv (m+1, INF);
    vector<bool> used (m+1, false);

    do {
      used[j0] = true;
      i0 = match[j0];
      j1 = -1;
      delta = INF;

      for(int j=1; j<=m; j++)
        if(!used[j]){
          TP cur = matrix[i0][j] - row[i0] - col[j];

          if( cur < minv[j] ) minv[j] = cur, way[j] = j0;
          if(minv[j] < delta) delta = minv[j], j1 = j;
        }

      for(int j=0; j<=m; j++)
        if(used[j]){
          row[match[j]] += delta,
          col[j] -= delta;
        }else
          minv[j] -= delta;

      j0 = j1;
    } while(match[j0]);


    do {
      j1 = way[j0];
      match[j0] = match[j1];
      j0 = j1;
    } while(j0);
  }

  return -col[0];
}

vector<pair<int, int>> getAssignment(int m){
  vector<pair<int, int>> ans;

  for(int i=1; i<=m; i++)
    ans.push_back(make_pair(match[i], i));

  return ans;
}
```

```
/**************************************
  Hungarian Algorithm - Assignment Problem
Algoritmo para o problema de atribui  o m nima.

Complexity: O(N^2 * M)

hungarian(int n, int m);  ->  Retorna o valor do custo m nimo
getAssignment(int m)      ->  Retorna a lista de pares <linha, Coluna> do
    Minimum Assignment

n -> N mero de Linhas // m -> N mero de Colunas

IMPORTANTE! O algoritmo   1-indexado
IMPORTANTE! O tipo padr o est  como int, para mudar para outro tipo
    altere | typedef <TIPO> TP; |
Extra: Para o problema da atribui  o m xima, apenas multiplique os
    elementos da matriz por -1
**************************************/
```

6.2 MO

```
const int BLOCK_SZ = 700;

struct Query{
  int l, r, idx;

  Query(int l, int r, int idx) : l(l), r(r), idx(idx) {}

  bool operator < (Query q) const {
    if(l / BLOCK_SZ != q.l / BLOCK_SZ) return l < q.l;
    return (l / BLOCK_SZ &1) ? ( r < q.r ) : (r > q.r );
  }
};

void add(int idx);
void remove(int idx);
int getAnswer();

vector<int> MO(vector<Query> &queries){
  vector<int> ans(queries.size());

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

  int L = 0, R = 0;
  add(0);

  for(auto [l, r, idx] : queries)
  {
    while(l < L) add(--L);
    while(r > R) add(++R);
    while(l > L) remove(L++);
    while(r < R) remove(R--);

    ans[idx] = getAnswer();
  }

  return ans;
}

/**************************************
Algoritmo de MO para query em range

Complexity: O( (N + Q) * SQRT(N) * F ) | F   a complexidade do Add e
    Remove

IMPORTANTE! Queries devem ter seus  ndices  (Idx) 0-indexados!

Modifique as opera  es de Add, Remove e GetAnswer de acordo com o
    problema.
BLOCK_SZ pode ser alterado para aproximadamente SQRT(MAX_N)
**************************************/
```

7 Strings

7.1 Hash

```
#define ll long long
const int MAXN = 1e6 + 5;

const ll MOD = 1e9 + 7; //WA? Muda o MOD e a base
const ll base = 153;

ll expBase[MAXN];

void precalc(){
  expBase[0] = 1;
  for(int i=1; i<MAXN; i++)
    expBase[i] = (expBase[i-1]*base)%MOD;
}

struct StringHash{
  vector<ll> hsh;

  StringHash(string &s){
    hsh = vector<ll>(s.size()+1, 0);
    for(int i=0; i<s.size(); i++)
      hsh[i+1] = ((hsh[i]*base) % MOD +s[i]) % MOD;
  }

  ll gethash(int l, int r){
    return (MOD + hsh[r+1] - (hsh[l]*expBase[r-l+1]) % MOD ) % MOD;
  }
};
/*******************************************************
String Hash
precalc()    -> O(N)
StringHash() -> O(|S|)
gethash()    -> O(1)

StringHash hash(s); -> Cria uma struct de StringHash para a string s
hash.gethash(l, r); -> Retorna o hash do intervalo L R da string (0-
    Indexado)

IMPORTANTE! Chamar precalc() no in cio do c digo

const ll MOD  = 131'807'699; -> Big Prime Number
const ll base = 127;         -> Random number larger than the Alphabet
*******************************************************/
```

7.2 Hash2

```
#define ll long long
const int MAXN = 1e6 + 5;

const ll MOD1 = 131'807'699;
const ll MOD2 = 1e9 + 9;
const ll base = 157;

ll expBase1[MAXN];
ll expBase2[MAXN];

void precalc(){
    expBase1[0] = expBase2[0] = 1;

  for(int i=1;i<MAXN;i++)
        expBase1[i] = ( expBase1[i-1]*base ) % MOD1,
        expBase2[i] = ( expBase2[i-1]*base ) % MOD2;
}

struct StringHash{
    vector<pair<ll,ll>> hsh;

    StringHash(string& s){     //!!! RUN PRECALC FIRST !!!
        hsh = vector<pair<ll,ll>> (s.size()+1, {0,0});

        for (int i=0;i<s.size();i++)
            hsh[i+1].first  = ( (hsh[i].first *base) % MOD1 + s[i] ) % MOD1
                ;
            hsh[i+1].second = ( (hsh[i].second*base) % MOD2 + s[i] ) % MOD2
                ;
    }

    ll gethash(int a,int b){
        ll h1 = (MOD1 + hsh[b+1].first  - ( hsh[a].first *expBase1[b-a+1] )
            % MOD1) % MOD1;
        ll h2 = (MOD2 + hsh[b+1].second - ( hsh[a].second*expBase2[b-a+1] )
            % MOD2) % MOD2;
        return (h1<<32LL) | h2;
    }
};

/*******************************************************
String Hash - Double Hash
precalc()    -> O(N)
StringHash() -> O(|S|)
gethash()    -> O(1)

StringHash hash(s); -> Cria o Hash da string s
hash.gethash(l, r); -> Hash [L,R] (0-Indexado)

IMPORTANTE! Chamar precalc() no in cio do c digo

Some Big Prime Numbers:
37'139'213
127'065'427
131'807'699
************************/
```

7.3 KMP

```
vector<int> pi(string &t){
  vector<int> p(t.size(), 0);

  for(int i=1, j=0; i<t.size(); i++)
  {
    while(j > 0  && t[j] != t[i]) j = p[j-1];

    if(t[j] == t[i]) j++;

    p[i] = j;
  }

  return p;
}

vector<int> kmp(string &s, string &t){
  vector<int> p = pi(t), occ;

  for(int i=0, j=0; i<s.size(); i++)
  {
    while( j > 0 && s[i] != t[j]) j = p[j-1];

    if(s[i]==t[j]) j++;

    if(j == t.size()) occ.push_back(i-j+1), j = p[j-1];
  }

  return occ;
}
/*************************
KMP -  K n u t h MorrisPratt  Pattern Searching
```

```
Complexity: O(|S|+|T|)

S -> String
T -> Pattern
***************************/
```

7.4 Manacher

```
vector<int> manacher(string &st){
  string s = "$_";
  for(char c : st){ s += c; s += "_"; }
  s += "#";

  int n = s.size()-2;

  vector<int> p(n+2, 0);
  int l=1, r=1;

  for(int i=1, j; i<=n; i++)
  {
    p[i] = max(0, min(r-i, p[l+r-i]) ); //atualizo o valor atual para o valor do palindromo espelho na string ou para o total que e st contido

    while( s[i-p[i]] == s[i+p[i]] ) p[i]++;

    if( i+p[i] > r ) l = i-p[i], r = i+p[i];
  }

  for(auto &x : p) x--; //o valor de p[i]   igual ao tamanho do palindromo + 1

  return p;
}

/******************
Manacher Algorithm
Find every palindrome in string
Complexidade: O(N)
******************/
```

7.5 trie

```
const int MAXS = 1e5 + 10;
const int sigma = 26;

int trie[MAXS][sigma], terminal[MAXS], z = 1;

void insert(string &p){
  int cur = 0;

  for(int i=0; i<p.size(); i++){
    int id = p[i] - 'a';

    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 = (p[i] - 'a');

    if(trie[cur][id] == -1) return 0;

    cur = trie[cur][id];
  }
  return terminal[cur];
}

void init(){
  memset(trie[0], -1, sizeof trie[0]);
  z = 1;
}
/******************
Trie - Arvore de Prefixos
insert(P) - O(|P|)
count(P)  - O(|P|)
MAXS  - Soma do tamanho de todas as Strings
sigma - Tamanho do alfabeto
*****************/
```

7.6 Z-Function

```
vector<int> Zfunction(string &s){ // O(N)
  int n = s.size();
  vector<int> z (n, 0);

  for(int i=1, l=0, r=0; i<n;  i++){
    if(i <= r) z[i] = min(z[i-l], r-i+1);

    while(z[i] + i < n && s[z[i]] == s[i+z[i]]) z[i]++;

    if(r < i+z[i]-1) l = i, r = i+z[i]-1;
  }

  return z;
}
```