Team Notebook

Fruta Fresca - UTN

September 3, 2024

Contents

1 Data structures

1.1 DSU & DSU rollbacks

```
struct UnionFind {
  int nsets;
  vector<int> f, setsz; // f[i] = parent of node i
  UnionFind(int n) : nsets(n), f(n, -1), setsz(n, 1) {}
  int comp(int x){return (f[x]==-1 ? x : f[x]=comp(f[x]));}
  bool join(int i, int j) {//returns true if already in same
    int a = comp(i), b = comp(j);
    if (a != b) {
      if (setsz[a] > setsz[b]) swap(a, b);
      f[a] = b; // big group (b) now represents small (a)
      nsets--, setsz[b] += setsz[a];
    }
    return a == b;
  }
};

struct dsu_save {
  int v, rnkv, u, rnku;
  dsu_save() {}
  dsu_save(int _v, int _rnkv, int _u, int _rnku)
      : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};
struct dsu_with_rollbacks {
  vector<int> p, rnk;
  int comps;
  stack<dsu_save> op;
  dsu_with_rollbacks() {}
  dsu_with_rollbacks(int n) {
    p.rsz(n), rnk.rsz(n);
    forn(i, n) { p[i] = i, rnk[i] = 0; }
    comps = n;
  }
  int find_set(int v){return(v==p[v]) ? v : find_set(p[v]);}
  bool unite(int v, int u) {
    v = find_set(v), u = find_set(u);
    if (v == u) return false;
    comps--;
    if (rnk[v] > rnk[u]) swap(v, u);
    op.push(dsu_save(v, rnk[v], u, rnk[u]));
    p[v] = u;
    if (rnk[u] == rnk[v]) rnk[u]++;
    return true;
  }
  void rollback() {
    if (op.empty()) return;
    dsu_save x = op.top();
    op.pop(), comps++;
    p[x.v] = x.v, rnk[x.v] = x.rnkv;
    p[x.u] = x.u, rnk[x.u] = x.rnku;
  }
};
```

1.2 Fenwick tree

```
struct FenwickTree {
  int N; // replace vector with unordered_map when "many 0s"
  vector<tipo> ft; // for more dims, make ft multi-dim
  FenwickTree(int n) : N(n), ft(n + 1) {}
  void upd(int i0, tipo v){//add v to i0th element (0-based)
    // add extra fors for more dimensions
    for (int i = i0 + 1; i <= N; i += i & -i) ft[i] += v;
  }
  tipo get(int i0) { // get sum of range [0,i0)
    tipo r = 0;     // add extra fors for more dimensions
    for (int i = i0; i; i -= i & -i) r += ft[i];
    return r;
  }
  tipo get_sum(int i0, int i1){//range sum [i0,i1) (0-based)
    return get(i1) - get(i0);
  }
};
```

1.3 Hash table

```
struct Hash { // similar logic for any other data type
  size_t operator()(const vector<int>& v) const {
    size_t s = 0;
    for (auto& e : v)
      s ^= hash<int>()(e) + 0x9e3779b9 + (s<<6) + (s>>2);
    return s;
  }
};
unordered_set<vector<int>, Hash> s; // map<key, val, Hash>
```

1.4 Indexed set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int,null_type,less<int>,rb_tree_tag,
 tree_order_statistics_node_update> indexed_set;
// find_by_order(i) returns iterator to the i-th elemnt
// order_of_key(k) returns position of the lb of k (0-index)
```

1.5 Link-cut tree

```
const int N_DEL = 0, N_VAL = 0; // neut for delta & values
inline int u_oper(int x, int y){ return x + y; } // upd oper
// query operation
inline int q_oper(int lval, int rval){ return lval + rval; }
// upd segment (maybe add 'inline')
int u_segm(int d, int len){return d==N_DEL?N_DEL:d*len;}
inline int u_delta(int d1, int d2){ // update delta
  if(d1==N_DEL) return d2;
  if(d2==N_DEL) return d1;
  return u_oper(d1, d2);

}
// apply delta (maybe add 'inline')
int a_delta(int v, int d){ return d==N_DEL?v:u_oper(d, v); }
struct node_t { // Splay tree
  int szi, n_val, t_val, d; bool rev;
  node_t *c[2], *p;
  node_t(int v):
        szi(1),n_val(v),t_val(v),d(N_DEL),rev(0),p(0) {
    c[0]=c[1]=0;
  }
  bool is_root(){return !p||(p->c[0]!=this&&p->c[1]!=this);}
  void push() {
    if(rev) {
      rev=0;swap(c[0],c[1]);forr(x,0,2)if(c[x])c[x]->rev^=1;
    }
    n_val = a_delta(n_val, d);
    t_val = a_delta(t_val, u_segm(d, szi));
    forr(x,0,2) if(c[x]) c[x]->d = u_delta(d, c[x]->d);
    d = N_DEL;
  }
  void upd();
};
typedef node_t* node;
int get_sz(node r) { return r ? r->szi : 0; }
int get_tree_val(node r) {
  return r ? a_delta(r->t_val, u_segm(r->d,r->szi)) : N_VAL;
}
void node_t::upd() {
  t_val=q_oper(q_oper(get_tree_val(c[0]),a_delta(n_val,d)),
             get_tree_val(c[1]));
  szi = 1 + get_sz(c[0]) + get_sz(c[1]);
}
void conn(node c, node p, int is_left) {
```

```
 if(c) c->p = p;
 if(is_left>=0) p->c[!is_left] = c;
}
void rotate(node x) {
  node p = x->p, g = p->p;
  bool gCh=p->is_root(), is_left = x==p->c[0];
  conn(x->c[is_left],p,is_left); conn(p,x,!is_left);
  conn(x,g,gCh?-1:(p==g->c[0])); p->upd();
}
void splay(node x) {
  while(!x->is_root()) {
    node p = x->p, g = p->p; if(!p->is_root()) g->push();
    p->push(); x->push();
    if(!p->is_root())rotate((x==p->c[0])==(p==g->c[0])?p:x);
    rotate(x);
  }
  x->push(); x->upd();
}
// Link-cut Tree
// Keep information of a tree (or forest) and allow to make
// many types of operations. Internally, each node will have
// at most 1 "preferred" child, and then, the tree can be
// seen as a set of independent "preferred" paths. Each of
// this paths is a list, represented with a splay tree,where
// the implicit key (for the BST) of each element is the
// depth of the corresponding node in the original tree (or
// forest). Also, each of these preferred paths, will know
// the preferred path of the father of the top-most node.

// Make the path from the root to 'x' to be a "preferred
// path", and also make 'x' to be the root of its splay tree
// (not the root of the original tree).
node expose(node x) {
  node last = 0;
  for(node y=x; y; y=y->p)
    splay(y), y->c[0] = last, y->upd(), last = y;
  splay(x);
  return last;
}
void make_root(node x) { expose(x); x->rev^=1; }
node get_root(node x) {
 expose(x); while(x->c[1]) x = x->c[1];
 splay(x); return x;
}
node lca(node x, node y) { expose(x); return expose(y); }
bool connected(node x, node y) {
  expose(x);expose(y);return x==y ? 1 : x->p!=0;
}
void link(node x, node y) { // makes x son of y
  make_root(x); x->p=y;
}
void cut(node x, node y){
  make_root(x); expose(y); y->c[1]->p=0; y->c[1]=0;
}
node father(node x){
 expose(x); node r = x->c[1];
 if(!r) return 0;
 while(r->c[0]) r = r->c[0];
 return r;
}
// cuts x from its father keeping tree root
void cut(node x){ expose(father(x)); x->p = 0; }
int query(node x, node y) {
  make_root(x); expose(y); return get_tree_val(y);
}
void update(node x, node y, int d){
 make_root(x); expose(y); y->d=u_delta(y->d,d);
}
node lift_rec(node x, int k) {
 if(!x) return 0;
 if(k == get_sz(x->c[0])) { splay(x); return x; }
 if(k < get_sz(x->c[0])) return lift_rec(x->c[0],k);
 return lift_rec(x->c[1], k-get_sz(x->c[0])-1);
}
node lift(node x, int k){ //k-th ancestor of x
  expose(x); return lift_rec(x,k);
}
int depth(node x) { // dist from x to tree root
  expose(x);return get_sz(x)-1;
}
```

1.6 Merge sort tree

```
typedef ii datain;   // data that goes into the DS
typedef int query;   // info related to a query
typedef bool dataout; // data that results from a query
struct DS {
  set<datain> s; // replace set with what you need
  void insert(const datain& x) {
    // modify this method according to problem
    // example below is disjoint intervals (union of ranges)
    datain xx = x; // copy to avoid changing original
    if (xx.fst >= xx.snd) return;
    auto at = s.lower_bound(xx);
    auto it = at;
    if (at != s.begin() && (--at)->snd >= xx.fst)
      xx.fst = at->fst, --it;
    for (;it!=s.end() && it->fst <= xx.snd; s.erase(it++))
      xx.snd = max(xx.snd, it->snd);
    s.insert(xx);
  }
  void get(const query& q, dataout& ans) {
    // modify this method according to problem
    // example below is "is there any range covering q?"
    set<datain>::iterator ite = s.ub(mp(q + 1, 0));
    if (ite != s.begin() && prev(ite)->snd > q) ans = true;
  }
};
struct MST {
  int sz;
  vector<DS> t;
  MST(int n) {
    sz = 1 << (32 - __builtin_clz(n));
    t = vector<DS>(2 * sz);
  }
  void insert(int i,int j, datain& x){insert(i,j,x,1,0,sz);}
  void insert(int i, int j, datain& x, int n, int a, int b){
    if (j <= a || b <= i) return;
    if (i <= a && b <= j) {
      t[n].insert(x);
      return;
    }
    // when want to update ranges that intersec with [i,j)
    // usually only on range-query + point-update problem
    // t[n].insert(x);
    int c = (a + b) / 2;
    insert(i, j, x, 2 * n, a, c);
    insert(i, j, x, 2 * n + 1, c, b);
  }
  void get(int i, int j, query& q, dataout& ans) {
    return get(i, j, q, ans, 1, 0, sz);
  }
  void get(int i, int j, query& q, dataout& ans,
          int n, int a, int b) {
    if (j <= a || b <= i) return;
    if (i <= a && b <= j) {
      t[n].get(q, ans);
      return;
    }
    // when want to get from ranges that intersec with [i,j)
    // usually only on point-query + range-update problem
    // t[n].get(q, ans);
    int c = (a + b) / 2;
    get(i, j, q, ans, 2 * n, a, c);
    get(i, j, q, ans, 2 * n + 1, c, b);
  }
}; // Use: 1- definir todo lo necesario en DS, 2- usar
```

1.7 Rope

```
#include <ext/rope>
using namespace __gnu_cxx;
rope<int> s;
// Sequence with O(logn) access, insert, erase any pos
// s.push_back(x)
// s.append(other_rope)
// s.insert(i,x)
// s.insert(i,other_rope) // insert rope r at position i
// s.erase(i,k) // erase subsequence [i,i+k)
// s.substr(i,k) // get new rope corresponding to [i,i+k)
// s[i] // get element (cannot modify)
// s.mutable_reference_at(i) // get element (allows modif)
// s.begin() and s.end() are const iterators
// (use mutable_begin(), mutable_end() to allow modif)
```

1.8 SegTree 2D

```
#define oper(x, y) max(x, y)
int n, m;
int a[MAXN][MAXN], st[2 * MAXN][2 * MAXN];
void build() { // O(n*m)
  forn(i, n) forn(j, m) st[i + n][j + m] = a[i][j];
  forn(i, n) dforn(j, m) // build st of row i+n
      st[i+n][j] = oper(st[i+n][j<<1], st[i+n][j<<1|1]);
  dforn(i, n) forn(j, 2 * m) // build st of ranges of rows
      st[i][j] = oper(st[i << 1][j], st[i << 1 | 1][j]);
}
void upd(int x, int y, int v) { // O(logn * logm)
  st[x + n][y + m] = v;
  for (int j = y+m; j>1; j>>=1)//upd ranges containing y+m
    st[x + n][j >> 1] = oper(st[x + n][j], st[x + n][j^1]);
  for (int i = x+n; i>1; i>>=1)//in each range with row x+n
    for (int j = y+m; j; j>>=1)//update the ranges with y+m
      st[i >> 1][j] = oper(st[i][j], st[i ^ 1][j]);
}
int query(int x0, int x1, int y0, int y1) {// O(logn * logm)
  int r = neutro;
  // start at the bottom and move up each time
  for (int i0 = x0+n, i1 = x1+n; i0 < i1; i0>>=1, i1>>=1) {
    int t[4], q = 0;
    // if whole segment of row i0 is included, move right
    if (i0 & 1) t[q++] = i0++;
    // if whole segment of row i1-1 is included, move left
    if (i1 & 1) t[q++] = --i1;
    forn(k,q) for(int j0=y0+m, j1=y1+m;j0<j1;j0>>=1,j1>>=1){
      if (j0 & 1) r = oper(r, st[t[k]][j0++]);
      if (j1 & 1) r = oper(r, st[t[k]][--j1]);
    }
  }
  return r;
}
```

1.9 SegTree dynamic

```
struct ST {
  int sz; vector<tipo> t;
  ST(int n) {
    sz = 1 << (32 - __builtin_clz(n));
    t = vector<tipo>(2 * sz, neutro);
  }
  tipo& operator[](int p) { return t[sz + p]; }
  void updall() {dforn(i,sz) t[i] = oper(t[2*i], t[2*i+1]);}
  tipo get(int i, int j) { return get(i, j, 1, 0, sz); }
  tipo get(int i, int j, int n, int a, int b) { // [i, j)
    if (j <= a || b <= i) return neutro;
    if (i <= a && b <= j) return t[n]; // n = node of [a,b)
    int c = (a + b) / 2;
    return oper(get(i,j, 2*n, a, c), get(i,j, 2*n+1, c, b));
  }
  void set(int p, tipo val) { // O(log n)
    p += sz;
    while (p > 0 && t[p] != val) {
      t[p] = val; p /= 2;
      val = oper(t[p * 2], t[p * 2 + 1]);
    }
  }
}; // Use: definir oper tipo neutro
```

1.10 SegTree implicit

```
struct ST { // Compressed segtree, works for any range
  ST *lc, *rc; tipo val; int L, R;
  ST(int l, int r, tipo x = neutro) {
    lc = rc = nullptr; L = l, R = r, val = x;
  }
  ST(int l, int r, ST* lp, ST* rp) {
    if (lp != nullptr && rp != nullptr && lp->L > rp->L)
      swap(lp, rp);
    lc = lp, rc = rp, L = l, R = r;
    val = oper(lp==nullptr ? neutro : lp->val,
               rp==nullptr ? neutro : rp->val);
  }
  // O(log(R-L)), parameter 'isnew' only needed when persist
  // return ST* for persistent
  void set(int p, tipo x, bool isnew = false) {
    // if(!isnew) {                      // for persist
    //   ST* newnode = new ST(L, R, lc, rc); // for persist
    //   return newnode->set(p, x, true); // for persist
    // }                                 // for persist
    if (L + 1 == R) {
      val = x;
      return; // 'return this;' for persistent
    }
    int m = (L + R) / 2;
    ST** c = p < m ? &lc : &rc;
    if (!*c) *c = new ST(p, p + 1, x);
    else if ((*c)->L <= p && p < (*c)->R) {
      // *c = (*c)->set(p,x); // for persist
      (*c)->set(p, x);      // NOT persist
    } else {
      int l = L, r = R;
      while ((p < m) == ((*c)->L < m)) {
        if (p < m) r = m; else l = m;
        m = (l + r) / 2;
      }
      *c = new ST(l, r, *c, new ST(p, p + 1, x));
    }
    val=oper(lc ? lc->val : neutro, rc ? rc->val : neutro);
    // return this; // for persistent
  }
  tipo get(int ql, int qr) { // O(log(R-L))
    if (qr <= L || R <= ql) return neutro;
    if (ql <= L && R <= qr) return val;
    return oper(lc ? lc->get(ql, qr) : neutro,
                rc ? rc->get(ql, qr) : neutro);
  }
}; // Usage: 1- RMQ st(MIN_INDEX, MAX_INDEX) 2- normally use
```

1.11 SegTree lazy

```
struct ST {
  int sz;
  vector<Elem> t;
  vector<Alt> dirty; // Alt and Elem could be diff types
  ST(int n) {
    sz = 1 << (32 - __builtin_clz(n));
    t = vector<Elem>(2 * sz, neutro);
    dirty = vector<Alt>(2 * sz, neutro2);
  }
  Elem& operator[](int p) { return t[sz + p]; }
  void updall() {dforn(i,sz) t[i] = oper(t[2*i], t[2*i+1]);}
  void push(int n, int a, int b) {//push dirt to child nodes
    if (dirty[n] != neutro2) { //n = node of range [a,b)
```

```
      t[n] += dirty[n] * (b - a); //CHANGE for your problem
      if (n < sz) {
        dirty[2*n] += dirty[n]; // CHANGE for your problem
        dirty[2*n+1] += dirty[n]; // CHANGE for your problem
      }
      dirty[n] = neutro2;
    }
  }
  Elem get(int i, int j, int n, int a, int b) { // O(lgn)
    if (j <= a || b <= i) return neutro;
    push(n, a, b); // adjust value before using it
    if (i <= a && b <= j) return t[n]; // n = node of [a,b)
    int c = (a + b) / 2;
    return oper(get(i,j, 2*n, a, c), get(i,j, 2*n+1, c, b));
  }
  Elem get(int i, int j) { return get(i, j, 1, 0, sz); }
  void update(Alt val, int i, int j, int n, int a, int b) {
    push(n, a, b);
    if (j <= a || b <= i) return;
    if (i <= a && b <= j) {
      dirty[n] += val; // CHANGE for your problem
      push(n, a, b);
      return;
    }
    int c = (a + b) / 2;
    update(val,i,j, 2*n, a,c), update(val,i,j, 2*n+1, c,b);
    t[n] = oper(t[2 * n], t[2 * n + 1]);
  }
  void update(Alt val, int i,int j){update(val,i,j,1,0,sz);}
}; // Use: operacion, neutros, Alt, Elem, uso de dirty
```

1.12 SegTree persistent

```
struct ST {
  int n;
  vector<tipo> st;
  vector<int> L, R;
  ST(int nn) : n(nn), st(1, neutro), L(1, 0), R(1, 0) {}
  int new_node(tipo v, int l = 0, int r = 0) {
    int id = sz(st); st.pb(v), L.pb(l), R.pb(r);
    return id;
  }
  int init(vector<tipo>& v, int l, int r) {
    if (l + 1 == r) return new_node(v[l]);
    int m = (l+r)/2, a = init(v, l, m), b = init(v, m, r);
    return new_node(oper(st[a], st[b]), a, b);
  }
  int update(int cur, int pos, tipo val, int l, int r) {
    int id = new_node(st[cur], L[cur], R[cur]);
    if (l + 1 == r) { st[id] = val; return id; }
    int m = (l + r) / 2, ASD; // MUST USE THE ASD!!!
    if(pos<m) ASD=update(L[id], pos, val, l, m), L[id]=ASD;
    else ASD = update(R[id], pos, val, m, r), R[id] = ASD;
    st[id] = oper(st[L[id]], st[R[id]]);
    return id;
  }
  tipo get(int cur, int from, int to, int l, int r) {
    if (to <= l || r <= from) return neutro;
    if (from <= l && r <= to) return st[cur];
    int m = (l + r) / 2;
    return oper(get(L[cur], from, to, l, m),
            get(R[cur], from, to, m, r));
  }
  int init(vector<tipo>& v) { return init(v, 0, n); }
  int update(int root, int pos, tipo val) {
    return update(root, pos, val, 0, n);
  }
  tipo get(int root, int from, int to) {
    return get(root, from, to, 0, n);
  }
}; // usage: ST st(n); root = st.init(v) (or root = 0);
```

1.13 SegTree static

```
struct RMQ { // LVL such that 2^LVL>n
  tipo vec[LVL][1 << (LVL + 1)];
  tipo& operator[](int p) { return vec[0][p]; }
  tipo get(int i, int j) { // intervalo [i,j) - O(1)
    int p = 31 - __builtin_clz(j - i);
    return min(vec[p][i], vec[p][j - (1 << p)]);
  }
  void build(int n) { // O(nlogn)
    int mp = 31 - __builtin_clz(n);
    forn(p, mp) forn(x, n - (1 << p)) vec[p + 1][x] =
        min(vec[p][x], vec[p][x + (1 << p)]);
  }
}; // insert data with []; call build; answer queries
```

1.14 Treap

```
// An array represented as a treap, where the "key" is the
// index. However, the key is not stored explicitly, but can
// be calculated as the sum of the sizes of the left child
// of the ancestors where the node is in the right subtree
// of it. (commented parts are specific to range sum queries
// and other problems)
typedef struct item* pitem;
struct item {
  int pr, cnt, val;
  bool rev; // for reverse operation
  int sum;  // for range query
  int add;  // for lazy prop
  pitem l, r, p; // p: ptr to parent, for getRoot
  item(int val) : pr(rng()), cnt(1), val(val),
                  rev(false), sum(val), add(0) {
    l = r = p = NULL;
  }
};
void push(pitem node) {
  if (node) {
    if (node->rev) { // for reverse operation
      swap(node->l, node->r);
      if (node->l) node->l->rev ^= true;
      if (node->r) node->r->rev ^= true;
      node->rev = false;
    }
    // for lazy prop
    node->val+=node->add, node->sum+=node->cnt*node->add;
    if (node->l) node->l->add += node->add;
    if (node->r) node->r->add += node->add;
    node->add = 0;
  }
}
int cnt(pitem t) { return t ? t->cnt : 0; }
// for range query
int sum(pitem t) { return t ? push(t), t->sum : 0; }
void upd_cnt(pitem t) {
  if (t) {
    t->cnt = cnt(t->l) + cnt(t->r) + 1;
    t->sum = t->val + sum(t->l) + sum(t->r);// for range sum
    if (t->l) t->l->p = t;               // for getRoot
    if (t->r) t->r->p = t;               // for getRoot
    t->p = NULL;                         // for getRoot
  }
}
// O(log), sz: wanted size for L
void split(pitem node, pitem& L, pitem& R, int sz) {
  if (!node) { L = R = 0; return; }
  push(node);
  if (sz<=cnt(node->l)) split(node->l,L,node->l,sz), R=node;
  else split(node->r, node->r, R,sz-1-cnt(node->l)), L=node;
  upd_cnt(node);
}
void merge(pitem& result, pitem L, pitem R) { // O(log)
  push(L), push(R);
  if (!L || !R) result = L ? L : R;
```

```
  else if (L->pr > R->pr) merge(L->r, L->r, R), result = L;
  else merge(R->l, L, R->l), result = R;
  upd_cnt(result);
}
void insert(pitem& node, pitem x, int pos){//0-index O(log)
  pitem l, r; split(node, l, r, pos);
  merge(l, l, x); merge(node, l, r);
}
void erase(pitem& node, int pos) { // 0-index O(log)
  if (!node) return;
  push(node);
  if (pos == cnt(node->l)) merge(node, node->l, node->r);
  else if (pos < cnt(node->l)) erase(node->l, pos);
  else erase(node->r, pos - 1 - cnt(node->l));
  upd_cnt(node);
}
void reverse(pitem& node, int L, int R) { //[L, R) O(log)
  pitem t1, t2, t3;
  split(node, t1, t2, L); split(t2, t2, t3, R - L);
  t2->rev ^= true;
  merge(node, t1, t2); merge(node, node, t3);
}
//[L, R) O(log), lazy add
void add(pitem& node, int L, int R, int x) {
  pitem t1, t2, t3;
  split(node, t1, t2, L); split(t2, t2, t3, R - L);
  t2->add += x;
  merge(node, t1, t2); merge(node, node, t3);
}
//[L, R) O(log), range query get
int get(pitem& node, int L, int R) {
  pitem t1, t2, t3;
  split(node, t1, t2, L); split(t2, t2, t3, R - L);
  push(t2); int ret = t2->sum;
  merge(node, t1, t2); merge(node, node, t3);
  return ret;
}
void push_all(pitem t) { // for getRoot
  if (t->p) push_all(t->p);
  push(t);
}
pitem getRoot(pitem t, int& pos){//get root & pos for node t
  push_all(t); pos = cnt(t->l);
  while (t->p) {
    pitem p = t->p;
    if (t == p->r) pos += cnt(p->l) + 1;
    t = p;
  }
  return t;
}
void output(pitem t) { // useful for debugging
  if (!t) return;
  push(t); output(t->l); cout << ' ' << t->val;output(t->r);
}
```

2 Flow

2.1 Dinic

```
struct Edge {
  int u, v;
  ll cap, flow;
  Edge() {}
  Edge(int uu, int vv, ll c):u(uu),v(vv),cap(c),flow(0) {}
};
struct Dinic {
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;
  Dinic(int n) : N(n), g(n), d(n), pt(n) {}
  void addEdge(int u, int v, ll cap) {
    if (u != v) {
      g[u].pb(sz(E)); E.pb({u, v, cap});
      g[v].pb(sz(E)); E.pb({v, u, 0});
    }
  }
  bool BFS(int S, int T) {
    queue<int> q({S}); fill(d.begin(),d.end(),N+1); d[S]=0;
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k : g[u]) {
        Edge& e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1)
          d[e.v] = d[e.u] + 1, q.push(e.v);
      }
    }
    return d[T] != N + 1;
  }
  ll DFS(int u, int T, ll flow = -1) {
    if (u == T || flow == 0) return flow;
    for (int& i = pt[u]; i < sz(g[u]); ++i) {
      Edge &e = E[g[u][i]], &oe = E[g[u][i] ^ 1]; // careful
      if (d[e.v] == d[e.u] + 1) {
        ll amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (ll pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }
  ll maxFlow(int S, int T) {//O(V^2*E), 1-nets: O(sqrt(V)*E)
    ll total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (ll flow = DFS(S, T)) total += flow;
    }
    return total;
  }
};
// If an edge with a min flow demand is part of a cycle,
// then the result is not guaranteed to be correct, it could
// result in false positives
struct DinicWithDemands {
  int N;
  vector<pair<Edge, ll>> E; // (normal dinic edge, min flow)
  Dinic dinic;
  DinicWithDemands(int n) : N(n), E(0), dinic(n + 2) {}
  void addEdge(int u, int v, ll cap, ll minFlow) {
    assert(minFlow <= cap);
    if (u != v) E.pb(mp(Edge(u, v, cap), minFlow));
  }
  ll maxFlow(int S, int T) { // normal Dinic complexity
    int SRC = N, SNK = N + 1;
    ll minFlowSum = 0;
    forall(e, E) { // force the min flow
      minFlowSum += e->snd;
      dinic.addEdge(SRC, e->fst.v, e->snd);
      dinic.addEdge(e->fst.u, SNK, e->snd);
      dinic.addEdge(e->fst.u,e->fst.v, e->fst.cap - e->snd);
    }
    dinic.addEdge(T, S, INF); // INF >= max possible flow
    ll flow = dinic.maxFlow(SRC, SNK);
    if (flow < minFlowSum) return -1; //no valid flow exists
    assert(flow == minFlowSum);
    //Go to valid state in the network satisfying min flows
    forn(i, sz(E)) {
      forn(j, 4) {
        assert(j%2 || dinic.E[6 * i + j].flow == E[i].snd);
        dinic.E[6*i + j].cap = dinic.E[6*i + j].flow = 0;
      }
      dinic.E[6*i + 4].cap += E[i].snd;
      dinic.E[6*i + 4].flow += E[i].snd;
```

```
    }
    forn(i, 2)
   dinic.E[6*sz(E)+i].cap = dinic.E[6*sz(E)+i].flow = 0;
    dinic.maxFlow(S, T); // Just finish the maxFlow now
    flow = 0;            // get the result manually
    forall(e, dinic.g[S]) flow += dinic.E[*e].flow;
    return flow;
  }
};
```

2.2 Hopcroft Karp

```
struct HopcroftKarp { // [0,n)->[0,m) (ids independent)
  int n, m;
  vector<vector<int>> g;
  vector<int> mt, mt2, ds;
  HopcroftKarp(int nn, int mm) : n(nn), m(mm), g(n) {}
  void add(int a, int b) { g[a].pb(b); }
  bool bfs() {
    queue<int> q;
    ds = vector<int>(n, -1);
    forn(i, n) if (mt2[i] < 0) ds[i] = 0, q.push(i);
    bool r = false;
    while (!q.empty()) {
      int x = q.front();
      q.pop();
      for (int y : g[x]) {
        if (mt[y] >= 0 && ds[mt[y]] < 0) {
          ds[mt[y]] = ds[x] + 1, q.push(mt[y]);
        } else if (mt[y] < 0) r = true;
      }
    }
    return r;
  }
  bool dfs(int x) {
    for (int y : g[x]) {
      if (mt[y]<0 || ds[mt[y]] == ds[x] + 1 && dfs(mt[y])) {
        mt[y] = x, mt2[x] = y;
        return true;
      }
    }
    ds[x] = 1 << 30;
    return false;
  }
  int mm() { // O(sqrt(V)*E)
    int r = 0;
    mt = vector<int>(m, -1);
    mt2 = vector<int>(n, -1);
    while (bfs()) forn(i, n) if (mt2[i] < 0) r += dfs(i);
    return r;
  }
};
```

2.3 Hungarian

```
bool zz(td x) { return abs(x) < 1e-9; } // use x==0 for ints
struct Hungarian {
  int n;
  vector<vector<td>> cs; // td usually double or ll
  vector<int> L, R;
  Hungarian(int N, int M)//for max make INF=0 & negate costs
      : n(max(N, M)), cs(n, vector<td>(n)), L(n), R(n) {
    forn(x, N) forn(y, M) cs[x][y] = INF;
  }
  void set(int x, int y, td c) { cs[x][y] = c; }
  td assign() { // O(n^3)
    int mat = 0;
    vector<td> ds(n), u(n), v(n);
    vector<int> dad(n), sn(n);
    forn(i,n) u[i]=*min_element(cs[i].begin(), cs[i].end());
    forn(j, n) {
      v[j] = cs[0][j] - u[0];
      forr(i, 1, n) v[j] = min(v[j], cs[i][j] - u[i]);
    }
    L = R = vector<int>(n, -1);
    forn(i, n) forn(j, n)
      if (R[j] == -1 && zz(cs[i][j] - u[i] - v[j])) {
        L[i] = j, R[j] = i, mat++; break;
    }
    for (; mat < n; mat++) {
      int s = 0, j = 0, i;
      while (L[s] != -1) s++;
      fill(dad.begin(), dad.end(), -1);
      fill(sn.begin(), sn.end(), 0);
      forn(k, n) ds[k] = cs[s][k] - u[s] - v[k];
      while (1) {
        j = -1;
        forn(k,n)if(!sn[k] && (j==-1 || ds[k]<ds[j])) j=k;
        sn[j] = 1, i = R[j];
        if (i == -1) break;
        forn(k, n) if (!sn[k]) {
          td new_ds = ds[j] + cs[i][k] - u[i] - v[k];
          if (ds[k] > new_ds) ds[k] = new_ds, dad[k] = j;
        }
      }
      forn(k, n) if (k != j && sn[k]) {
        td w = ds[k] - ds[j];
        v[k] += w, u[R[k]] -= w;
      }
      u[s] += ds[j];
      while (dad[j] >= 0) {
        int d = dad[j]; R[j] = R[d], L[R[j]] = j, j = d;
      }
      R[j] = s, L[s] = j;
    }
    td ret = 0; forn(i, n) ret += cs[i][L[i]];
    return ret;
  }
};
```

2.4 MCMF

```
struct edge {
  int u, v;
  tf cap, flow; // tf usually ll or double
  tc cost; // tc usually ll or double
  tf rem() { return cap - flow; }
};
struct MCMF {
  vector<edge> e; vector<vector<int>> g;
  vector<tc> dist; vector<tf> vcap; vector<int> pre;
  tc minCost; tf maxFlow;
  // tf wantedFlow; // Use it for fixed flow
  MCMF(int n) : g(n), vcap(n), dist(n), pre(n) {}
  void addEdge(int u, int v, tf cap, tc cost) {
    g[u].pb(sz(e)), e.pb({u, v, cap, 0, cost});
    g[v].pb(sz(e)), e.pb({v, u, 0, 0, -cost});
  }
  void run(int s, int t) { // O(n*m*min(flow,n*m)) or faster
    vector<bool> inq(sz(g));
    maxFlow = minCost = 0; // result will be here
    while (1) {
      fill(vcap.begin(), vcap.end(), 0), vcap[s] = INF_FLOW;
      fill(dist.begin(), dist.end(), INF_COST), dist[s] = 0;
      fill(pre.begin(), pre.end(), -1), pre[s] = 0;
      queue<int> q; q.push(s), inq[s] = true;
      while (sz(q)) { // Fast bellman-ford
        int u = q.front(); q.pop(), inq[u] = false;
        for (auto eid : g[u]) {
          edge& E = e[eid];
          if (E.rem() && dist[E.v] > dist[u] + E.cost) {
            dist[E.v] = dist[u] + E.cost; pre[E.v] = eid;
            vcap[E.v] = min(vcap[u], E.rem());
            if (!inq[E.v]) q.push(E.v), inq[E.v] = true;
          }
        }
      }
```

```
      if (pre[t] == -1) break;
      tf flow = vcap[t];
      // flow = min(flow, wantedFlow - maxFlow);//fixed flow
      maxFlow += flow; minCost += flow * dist[t];
      for (int v = t; v != s; v = e[pre[v]].u)
        e[pre[v]].flow += flow, e[pre[v] ^ 1].flow -= flow;
      // if(maxFlow == wantedFlow) break; // for fixed flow
    }
  }
};
```

3 Geometry

3.1 All point pairs

```
// after each step() execution pt is sorted by dot product
// of the event. O(n*n*log(n*n)), must add id, u, v to pto
struct all_point_pairs {
  vector<pto> pt, ev; vector<int> idx; int cur_step;
  all_point_pairs(vector<pto> pt_) : pt(pt_) {
    idx = vector<int>(sz(pt));
    forn(i, sz(pt)) forn(j, sz(pt)) if (i != j) {
      pto p = pt[j] - pt[i];
      p.u = pt[i].id, p.v = pt[j].id;
      ev.pb(p);
    }
    sort(ev.begin(), ev.end(), cmp(pto(0, 0), pto(1, 0)));
    pto start(ev[0].y, -ev[0].x);
    sort(pt.begin(), pt.end(),
         [&](pto& u, pto& v) { return u*start < v*start; });
    forn(i, sz(idx)) idx[pt[i].id] = i;
    cur_step = 0;
  }
  bool step() {
    if (cur_step >= sz(ev)) return false;
    int u = ev[cur_step].u, v = ev[cur_step].v;
    swap(pt[idx[u]], pt[idx[v]]);
    swap(idx[u], idx[v]);
    cur_step++;
    return true;
  }
};
```

3.2 Circle

```
#define sqr(a) ((a) * (a))
pto perp(pto a) { return pto(-a.y, a.x); }
line bisector(pto a, pto b) {
  line l = line(a, b); pto m = (a + b) / 2;
  return line(-l.b, l.a, -l.b * m.x + l.a * m.y);
}
struct circle {
  pto o; T r;
  circle() {}
  circle(pto a, pto b, pto c) {
    o = bisector(a, b).inter(bisector(b, c)); r = o.dist(a);
  }
  bool inside(pto p) {return (o-p).norm_sq() <= r*r + EPS;}
  bool inside(circle c) { // this inside of c
    T d=(o-c.o).norm_sq(); return d <= (c.r-r)*(c.r-r)+EPS;
  }
  // circle containing p1 and p2 with radius r
  // swap p1, p2 to get snd solution
  circle* circle2PtoR(pto a, pto b, T r_) {
    ld d2 = (a - b).norm_sq(), det = r_*r_ / d2-ld(0.25);
    if (det < 0) return nullptr;
    circle* ret = new circle();
    ret->o = (a + b) / ld(2) + perp(b - a) * sqrt(det);
    ret->r = r_;
    return ret;
  }
  pair<pto, pto> tang(pto p) {
    pto m = (p + o) / 2;
    ld d = o.dist(m);
    ld a = r * r / (2 * d);
    ld h = sqrtl(r * r - a * a);
    pto m2 = o + (m - o) * a / d;
    pto per = perp(m - o) / d;
    return make_pair(m2 - per * h, m2 + per * h);
  }
  vector<pto> inter(line l) {
    ld a = l.a, b = l.b, c = l.c - l.a*o.x - l.b*o.y;
    pto xy0 = pto(a*c / (a*a + b*b), b*c / (a*a + b*b));
    if (c*c > r*r*(a*a + b*b) + EPS) { return {};
    } else if (abs(c*c - r*r*(a*a + b*b)) < EPS) {
      return {xy0 + o};
    } else {
      ld m = sqrtl((r*r - c*c / (a*a + b*b)) / (a*a + b*b));
      pto p1 = xy0 + (pto(-b, a) * m);
      pto p2 = xy0 + (pto(b, -a) * m);
      return {p1 + o, p2 + o};
    }
  }
  vector<pto> inter(circle c) {
    line l;
    l.a = o.x - c.o.x;
    l.b = o.y - c.o.y;
    l.c = (sqr(c.r) - sqr(r) + sqr(o.x) - sqr(c.o.x) +
           sqr(o.y) - sqr(c.o.y)) / 2.0;
    return (*this).inter(l);
  }
  ld inter_triangle(pto a, pto b) { //area of inter with oab
    if (abs((o - a) ^ (o - b)) <= EPS) return 0.;
    vector<pto> q = {a}, w = inter(line(a, b));
    if(sz(w)==2) forn(i,sz(w)) if((a-w[i])*(b-w[i]) < -EPS)
      q.pb(w[i]);
    q.pb(b);
    if (sz(q) == 4 && (q[0] - q[1]) * (q[2] - q[1]) > EPS)
      swap(q[1], q[2]);
    ld s = 0;
    forn(i, sz(q) - 1) {
      if (!inside(q[i]) || !inside(q[i + 1])) {
        s += r * r * angle((q[i] - o), q[i + 1] - o) / T(2);
      } else s += abs((q[i] - o) ^ (q[i + 1] - o) / 2);
    }
    return s;
  }
};
vector<ld> inter_circles(vector<circle> c) {
  // r[k]: area covered by at least k circles
  vector<ld> r(sz(c) + 1);
  forn(i, sz(c)) { // O(n^2 log n) (high constant)
    int k = 1; cmp s(c[i].o, pto(1, 0));
    vector<pair<pto, int>> p = {
        {c[i].o + pto(1, 0) * c[i].r, 0},
        {c[i].o - pto(1, 0) * c[i].r, 0}};
    forn(j, sz(c)) if (j != i) {
      bool b0 = c[i].inside(c[j]), b1 = c[j].inside(c[i]);
      if (b0 && (!b1 || i < j)) k++;
      else if (!b0 && !b1) {
        vector<pto> v = c[i].inter(c[j]);
        if (sz(v) == 2) {
          p.pb({v[0], 1}); p.pb({v[1], -1});
          if (s(v[1], v[0])) k++;
        }
      }
    }
    sort(p.begin(), p.end(),
         [&](pair<pto, int> a, pair<pto, int> b) {
          return s(a.fst, b.fst);
         });
    forn(j, sz(p)) {
      pto p0 = p[j ? j - 1 : sz(p) - 1].fst, p1 = p[j].fst;
      ld a = angle(p0 - c[i].o, p1 - c[i].o);
      r[k] += (p0.x - p1.x) * (p0.y + p1.y) / ld(2) +
              c[i].r * c[i].r * (a - sinl(a)) / ld(2);
```

```
      k += p[j].snd;
    }
  }
  return r;
}
```

3.3 Convex hull dynamic

```
struct semi_chull {
  set<pto> pt; // maintains semi chull without collinears
  // if you want collinears, make changes commented below
  bool check(pto p) {
    if (pt.empty()) return false;
    if (*pt.rbegin() < p) return false;
    if (p < *pt.begin()) return false;
    auto it = pt.lower_bound(p);
    if (it->x == p.x) return p.y <= it->y; // change?
    pto b = *it;
    pto a = *prev(it);
    return ((b - p) ^ (a - p)) + EPS >= 0; // change?
  }
  void add(pto p) {
    if (check(p)) return;
    pt.erase(p);
    pt.insert(p);
    auto it = pt.find(p);
    while (true) {
      if (next(it) == pt.end() ||
          next(next(it)) == pt.end()) break;
      pto a = *next(it), b = *next(next(it));
      if (((b - a) ^ (p - a)) + EPS >= 0) { // change?
        pt.erase(next(it));
      } else break;
    }
    it = pt.find(p);
    while (true) {
      if (it == pt.begin() || prev(it) == pt.begin()) break;
      pto a = *prev(it), b = *prev(prev(it));
      if (((b - a) ^ (p - a)) - EPS <= 0) { // change?
        pt.erase(prev(it));
      } else break;
    }
  }
};
struct CHD {
  semi_chull sup, inf;
  void add(pto p) { sup.add(p), inf.add(p * (-1)); }
  bool check(pto p){return sup.check(p)&&inf.check(p*(-1));}
};
```

3.4 Convex hull trick

```
struct CHT {
  deque<pto> h; T f = 1, pos;
   // min_=1 for min queries
  CHT(bool min_ = 0) : f(min_ ? 1 : -1), pos(0) {}
  void add(pto p) { // O(1), pto(m,b) <=> y = mx + b
    p = p * f;
    if (h.empty()) { h.pb(p); return; }
    // p.x should be the lower/greater hull x
    assert(p.x <= h[0].x || p.x >= h.back().x);
    if (p.x <= h[0].x) {
      while(sz(h)>1&&h[0].left(p,h[1])) h.pop_front(),pos--;
      h.push_front(p), pos++;
    } else {
      while(sz(h) > 1 && h[sz(h)-1].left(h[sz(h)-2], p))
        h.pop_back();
      h.pb(p);
    }
    pos = min(max(T(0), pos), T(sz(h) - 1));
  }
  T get(T x) {
    pto q = {x, 1};
    // O(log) query for unordered x
    int L = 0, R = sz(h) - 1, M;
    while (L < R) {
      M = (L + R) / 2;
      if (h[M + 1] * q <= h[M] * q) L = M + 1;
      else R = M;
    }
    return h[L] * q * f;
    // O(1) query for ordered x
    while (pos > 0 && h[pos-1]*q < h[pos]*q) pos--;
    while (pos < sz(h)-1 && h[pos+1]*q < h[pos]*q) pos++;
    return h[pos] * q * f;
  }
};
```

3.5 Convex hull

```
// chull in CCW, make left>=0 todelete collinears
vector<pto> CH(vector<pto>& p) {
  if (sz(p) < 3) return p; // edge case, keep line or point
  vector<pto> ch;
  sort(p.begin(), p.end());
  forn(i, sz(p)) { // lower hull
    while(sz(ch)>=2 && ch[sz(ch)-1].left(ch[sz(ch)-2],p[i]))
      ch.pop_back();
    ch.pb(p[i]);
  }
  ch.pop_back();
  int k = sz(ch);
  dforn(i, sz(p)) { // upper hull
    while(sz(ch)>=k+2&&ch[sz(ch)-1].left(ch[sz(ch)-2],p[i]))
      ch.pop_back();
    ch.pb(p[i]);
  }
  ch.pop_back();
  return ch;
}
```

3.6 Halfplane

```
struct halfplane { // left half plane
  pto u, uv; int id; ld angle;
  halfplane() {}
  halfplane(pto u_, pto v_)
      : u(u_), uv(v_ - u_), angle(atan2l(uv.y, uv.x)) {}
  bool operator<(halfplane h) const{return angle < h.angle;}
  bool out(pto p) { return (uv ^ (p - u)) < -EPS; }
  pto inter(halfplane& h) {
    T alpha = ((h.u - u) ^ h.uv) / (uv ^ h.uv);
    return u + (uv * alpha);
  }
};
vector<pto> intersect(vector<halfplane> h) {
  pto box[4]={{INF,INF},{-INF,INF},{-INF,-INF},{INF,-INF}};
  forn(i, 4) h.pb(halfplane(box[i], box[(i + 1) % 4]));
  sort(h.begin(), h.end()); deque<halfplane> dq; int len=0;
  forn(i, sz(h)) {
    while (len > 1 && h[i].out(dq[len-1].inter(dq[len-2])))
      dq.pop_back(), len--;
    while (len > 1 && h[i].out(dq[0].inter(dq[1])))
      dq.pop_front(), len--;
    if (len > 0 && abs(h[i].uv ^ dq[len - 1].uv) <= EPS) {
      if(h[i].uv * dq[len-1].uv < 0.) return vector<pto>();
      if(h[i].out(dq[len-1].u)) dq.pop_back(), len--;
      else continue;
    }
    dq.pb(h[i]); len++;
  }
  while (len > 2 && dq[0].out(dq[len-1].inter(dq[len-2])))
    dq.pop_back(), len--;
  while (len > 2 && dq[len - 1].out(dq[0].inter(dq[1])))
```

```
    dq.pop_front(), len--;
  if (len < 3) return vector<pto>();
  vector<pto> inter;
  forn(i, len) inter.pb(dq[i].inter(dq[(i + 1) % len]));
  return inter;
}
```

3.7 KD tree

```
bool cmpx(pto a, pto b) { return a.x + EPS < b.x; }
bool cmpy(pto a, pto b) { return a.y + EPS < b.y; }
struct kd_tree {
  pto p; T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF;
  kd_tree *l, *r;
  T distance(pto q) {
    T x = min(max(x0, q.x), x1), y = min(max(y0, q.y), y1);
    return (pto(x, y) - q).norm_sq();
  }
  kd_tree(vector<pto>&& pts) : p(pts[0]) {
    l = nullptr, r = nullptr;
    forn(i, sz(pts)) {
      x0 = min(x0, pts[i].x), x1 = max(x1, pts[i].x);
      y0 = min(y0, pts[i].y), y1 = max(y1, pts[i].y);
    }
    if (sz(pts) > 1) {
      sort(pts.begin(), pts.end(), x1-x0>=y1-y0?cmpx:cmpy);
      int m = sz(pts) / 2;
      l = new kd_tree({pts.begin(), pts.begin() + m});
      r = new kd_tree({pts.begin() + m, pts.end()});
    }
  }
  void nearest(pto q,int k,priority_queue<pair<T,pto>>&ret){
    if (l == nullptr) {
      // if(p == q) return; // avoid query point as answer
      ret.push({(q - p).norm_sq(), p});
      while (sz(ret) > k) ret.pop();
      return;
    }
    kd_tree *al = l, *ar = r;
    T bl = l->distance(q), br = r->distance(q);
    if (bl > br) swap(al, ar), swap(bl, br);
    al->nearest(q, k, ret);
    if (br < ret.top().fst) ar->nearest(q, k, ret);
    while (sz(ret) > k) ret.pop();
  }
  priority_queue<pair<T, pto>> nearest(pto q, int k) {
    priority_queue<pair<T, pto>> ret;
    forn(i, k) ret.push({INF * INF, pto(INF, INF)});
    nearest(q, k, ret);
    return ret;
  }
};
```

3.8 Li-Chao tree

```
struct line {
  T m, b;
  line() {}
  line(T m_, T b_) : m(m_), b(b_) {}
  T f(T x) { return m * x + b; }
  line operator+(line l) { return line(m + l.m, b + l.b); }
  line operator*(T k) { return line(m * k, b * k); }
};
struct li_chao {
  vector<line> cur, add; vector<int> L, R;
  T f, minx, maxx; line identity; int cnt;
  void new_node(line cur_, int l = -1, int r = -1) {
    cur.pb(cur_); add.pb(line(0, 0)); L.pb(l),R.pb(r),cnt++;
  }
  li_chao(bool min_, T minx_, T maxx_) { // for max: min_=0
    f = min_ ? 1 : -1; identity = line(0, INF);
    minx = minx_; maxx = maxx_; cnt = 0;
    new_node(identity); // root id is 0
  }
  // only when "adding" lines lazily
  void apply(int id, line to_add_) {
    add[id] = add[id] + to_add_;
    cur[id] = cur[id] + to_add_;
  }
  void push_lazy(int id) { // MUST use (even when not lazy)
    if (L[id] == -1) new_node(identity), L[id] = cnt - 1;
    if (R[id] == -1) new_node(identity), R[id] = cnt - 1;
    // code below only needed when lazy ops are needed
    apply(L[id], add[id]); apply(R[id], add[id]);
    add[id] = line(0, 0);
  }
  // only when "adding" lines lazily
  void push_line(int id, T tl, T tr) {
    T m = (tl + tr) / 2;
    insert_line(L[id], cur[id], tl, m);
    insert_line(R[id], cur[id], m, tr);
    cur[id] = identity;
  }
  // O(log), for persistent return int
  void insert_line(int id, line new_line, T l, T r) {
    T m = (l + r) / 2;
    bool lef = new_line.f(l) < cur[id].f(l);
    bool mid = new_line.f(m) < cur[id].f(m);
    // line to_push = new_line, to_keep = cur[id]; //persist
    // if(mid) swap(to_push,to_keep);          //persist
    if (mid) swap(new_line, cur[id]);     // NOT persist?
    if (r - l == 1) {
      // new_node(to_keep); return cnt-1;     // persist
      return;                               // NOT persist
    }
    push_lazy(id);
    if (lef != mid) {
      // int lid = insert_line(L[id],to_push,l,m); //persist
      // new_node(to_keep, lid, R[id]);         //persist
      // return cnt-1;                          //persist
      insert_line(L[id], new_line, l, m); // NOT persist
    } else {
      // int rid = insert_line(R[id],to_push,m,r); //persist
      // new_node(to_keep, L[id], rid);         //persist
      // return cnt-1;                          //persist
      insert_line(R[id], new_line, m, r); // NOT persist
    }
  }
  void insert_line( int id, line new_line){//persist ret int
    insert_line(id, new_line * f, minx, maxx);
  }
  // O(log^2) doesn't support persistence
  void insert_segm(int id,line new_line,T l,T r,T tl,T tr) {
    if (tr <= l || tl >= r || tl >= tr || l >= r) return;
    if (tl >= l && tr <= r) {
      insert_line(id, new_line, tl, tr); return;
    }
    push_lazy(id); T m = (tl + tr) / 2;
    insert_segm(L[id], new_line, l, r, tl, m);
    insert_segm(R[id], new_line, l, r, m, tr);
  }
  void insert_segm(int id, line new_line, T l, T r) {//[l,r)
    insert_segm(id, new_line * f, l, r, minx, maxx);
  }
  // O(log^2) doesn't support persistence
  void add_line(int id, line to_add_, T l, T r, T tl, T tr){
    if (tr <= l || tl >= r || tl >= tr || l >= r) return;
    if (tl >= l && tr <= r) { apply(id, to_add_); return; }
    push_lazy(id);
    push_line(id, tl, tr); // comment if insert isn't used
    T m = (tl + tr) / 2;
    add_line(L[id], to_add_, l, r, tl, m);
    add_line(R[id], to_add_, l, r, m, tr);
  }
  void add_line(int id, line to_add_, T l, T r) {
    add_line(id, to_add_ * f, l, r, minx, maxx);
  }
  T get(int id, T x, T tl, T tr) { // O(log)
```

```
    if (tl + 1 == tr) return cur[id].f(x);
    push_lazy(id); T m = (tl + tr) / 2;
    if(x<m) return min(cur[id].f(x), get(L[id], x, tl, m));
    else return min(cur[id].f(x), get(R[id], x, m, tr));
  }
  T get(int id, T x) { return get(id, x, minx, maxx) * f; }
};
```

3.9 Line

```
int sgn(T x) { return x < 0 ? -1 : !!x; }
struct line {
  T a, b, c; // Ax+By=C
  line() {}
  line(T a_, T b_, T c_) : a(a_), b(b_), c(c_) {}
  // TO DO: check negative C (multiply everything by -1)
  line(pto u, pto v):a(v.y-u.y),b(u.x-v.x),c(a*u.x+b*u.y){}
  int side(pto v) { return sgn(a * v.x + b * v.y - c); }
  bool inside(pto v) {return abs(a*v.x + b*v.y - c) <= EPS;}
  bool parallel(line v) {return abs(a*v.b - v.a*b) <= EPS;}
  pto inter(line v) {
    T det = a * v.b - v.a * b;
    if (abs(det) <= EPS) return pto(INF, INF);
    return pto(v.b * c - b * v.c, a * v.c - v.a * c) / det;
  }
};
```

3.10 Point

```
struct pto {
  T x, y;
  pto() : x(0), y(0) {}
  pto(T _x, T _y) : x(_x), y(_y) {}
  pto operator+(pto b) { return pto(x + b.x, y + b.y); }
  pto operator-(pto b) { return pto(x - b.x, y - b.y); }
  pto operator+(T k) { return pto(x + k, y + k); }
  pto operator*(T k) { return pto(x * k, y * k); }
  pto operator/(T k) { return pto(x / k, y / k); }
  // dot product
  T operator*(pto b) { return x * b.x + y * b.y; }
  // module of cross product, a^b>0 if angle_cw(u,v)<180
  T operator^(pto b) { return x * b.y - y * b.x; }
  // vector projection of this above b
  pto proj(pto b) { return b * ((*this) * b) / (b * b); }
  T norm_sq() { return x * x + y * y; }
  ld norm() { return sqrtl(x * x + y * y); }
  ld dist(pto b) { return (b - (*this)).norm(); }
  // rotate by theta rads CCW w.r.t. origin (0,0)
  pto rotate(T ang) {
    return pto(x * cosl(ang) - y * sinl(ang),
               x * sinl(ang) + y * cosl(ang));
  }
  // true if this is at the left side of line ab
  bool left(pto a, pto b){return ((a-*this)^(b-*this)) > 0;}
  bool operator<(const pto& b) const {
    return x< b.x-EPS || (abs(x-b.x) <= EPS && y< b.y-EPS);
  }
  bool operator==(pto b) {
    return abs(x - b.x) <= EPS && abs(y - b.y) <= EPS;
  }
};
ld angle(pto a, pto o, pto b) {
  pto oa = a - o, ob = b - o; return atan2l(oa^ob, oa*ob);
}
ld angle(pto a, pto b) { // smallest angle bewteen a and b
  ld cost = (a * b) / a.norm() / b.norm();
  return acosl(max(ld(-1.), min(ld(1.), cost)));
}
```

3.11 Polygon

```
struct poly {
  vector<pto> pt;
  poly() {}
  poly(vector<pto> pt_) : pt(pt_) {}
  void delete_collinears() { // delete collinear points
    deque<pto> nxt; int len = 0;
    forn(i, sz(pt)) {
      if (len > 1 && abs((pt[i] - nxt[len-2]) ^
          (nxt[len-1] - nxt[len-2])) <= EPS)
        nxt.pop_back(), len--;
      nxt.pb(pt[i]); len++;
    }
    if (len > 2 && abs((nxt[1] - nxt[len - 1]) ^
                       (nxt[0] - nxt[len - 1])) <= EPS)
      nxt.pop_front(), len--;
    if (len > 2 && abs((nxt[len - 1] - nxt[len - 2]) ^
                       (nxt[0] - nxt[len - 2])) <= EPS)
      nxt.pop_back(), len--;
    pt.clear();
    forn(i, sz(nxt)) pt.pb(nxt[i]);
  }
  void normalize() {
    delete_collinears();
    if (pt[2].left(pt[0], pt[1]))
      reverse(pt.begin(), pt.end()); // make it CW
    int n = sz(pt), pi = 0;
    forn(i, n) if (pt[i].x < pt[pi].x ||
                   (pt[i].x == pt[pi].x &&
                    pt[i].y < pt[pi].y)) pi = i;
    rotate(pt.begin(), pt.begin() + pi, pt.end());
  }
  bool is_convex() { // delete collinear points first
    int N = sz(pt);
    if (N < 3) return false;
    bool isLeft = pt[0].left(pt[1], pt[2]);
    forr(i,1,sz(pt))if(pt[i].left(pt[(i+1)%N],
                                 pt[(i+2)%N]) != isLeft)
      return false;
    return true;
  }
  // for convex or concave polygons
  // excludes boundaries, check it manually
  bool inside(pto p) { // O(n)
    bool c = false;
    forn(i, sz(pt)) {
      int j = (i + 1) % sz(pt);
      if ((pt[j].y > p.y) != (pt[i].y > p.y) && (p.x <
          (pt[i].x - pt[j].x) * (p.y - pt[j].y) /
          (pt[i].y - pt[j].y) + pt[j].x))
        c = !c;
    }
    return c;
  }
  bool inside_convex(pto p) { // O(lg(n)) normalize first
    if (p.left(pt[0], pt[1]) || p.left(pt[sz(pt)-1], pt[0]))
      return false;
    int a = 1, b = sz(pt) - 1;
    while (b - a > 1) {
      int c = (a + b) / 2;
      if (!p.left(pt[0], pt[c])) a = c;
      else b = c;
    }
    return !p.left(pt[a], pt[a + 1]);
  }
  // cuts this along line ab and return the left side
  // (swap a, b for the right one)
  poly cut(pto a, pto b) { // O(n)
    vector<pto> ret;
    forn(i, sz(pt)) {
      ld left1 = (b - a) ^ (pt[i] - a),
        left2 = (b - a) ^ (pt[(i + 1) % sz(pt)] - a);
      if (left1 >= 0) ret.pb(pt[i]);
      if (left1 * left2 < 0)
        ret.pb(line(pt[i], pt[(i + 1) % sz(pt)])
                   .inter(line(a, b)));
```

```
  }
  return poly(ret);
}
// cuts this with line ab and returns the range [from, to]
// that is strictly on the left side (circular indexes)
ii cut(pto u, pto v) { // O(log(n)) for convex polygons
  int n = sz(pt); pto dir = v - u;
  int L = farthest(pto(dir.y, -dir.x));
  int R = farthest(pto(-dir.y, dir.x));
  if (!pt[L].left(u, v)) swap(L, R);
  if (!pt[L].left(u, v)) return mp(-1, -1); // no cut
  ii ans; int l = L, r = L > R ? R + n : R;
  while (l < r) {
    int med = (l + r + 1) / 2;
    if (pt[med >= n ? med - n : med].left(u, v)) l = med;
    else r = med - 1;
  }
  ans.snd = l >= n ? l - n : l;
  l = R, r = L < R ? L + n : L;
  while (l < r) {
    int med = (l + r) / 2;
    if (!pt[med >= n ? med-n : med].left(u, v)) l = med+1;
    else r = med;
  }
  ans.fst = l >= n ? l - n : l;
  return ans;
}
// addition of convex polygons
poly minkowski(poly p) { // O(n+m) n=|this|,m=|p|
  this->normalize(); p.normalize();
  vector<pto> a = (*this).pt, b = p.pt;
  a.pb(a[0]); a.pb(a[1]); b.pb(b[0]); b.pb(b[1]);
  vector<pto> sum; int i = 0, j = 0;
  while (i < sz(a) - 2 || j < sz(b) - 2) {
    sum.pb(a[i] + b[j]);
    T cross = (a[i + 1] - a[i]) ^ (b[j + 1] - b[j]);
    if (cross <= 0 && i < sz(a) - 2) i++;
    if (cross >= 0 && j < sz(b) - 2) j++;
  }
  return poly(sum);
}
pto farthest(pto v) { // O(log(n)) for convex polygons
  if (sz(pt) < 10) {
    int k = 0;
    forr(i,1,sz(pt)) if (v * (pt[i] - pt[k]) > EPS) k = i;
    return pt[k];
  }
  pt.pb(pt[0]); pto a = pt[1] - pt[0];
  int s = 0, e = sz(pt) - 1, ua = v * a > EPS;
  if (!ua && v * (pt[sz(pt) - 2] - pt[0]) <= EPS) {
    pt.pop_back(); return pt[0];
  }
  while (1) {
    int m = (s + e) / 2;
    pto c = pt[m + 1] - pt[m];
    int uc = v * c > EPS;
    if (!uc && v * (pt[m - 1] - pt[m]) <= EPS) {
      pt.pop_back(); return pt[m];
    }
    if (ua && (!uc || v * (pt[s] - pt[m]) > EPS)) e = m;
    else if(ua||uc || v*(pt[s]-pt[m])>=-EPS)s=m,a=c,ua=uc;
    else e = m;
    assert(e > s + 1);
  }
}
ld inter_circle(circle c) { //area of inter with circle
  ld r = 0.;
  forn(i, sz(pt)) {
    int j = (i + 1) % sz(pt);
    ld w = c.inter_triangle(pt[i], pt[j]);
    if (((pt[j] - c.o) ^ (pt[i] - c.o)) > 0) r += w;
    else r -= w;
  }
  return abs(r);
}
// area ellipse = M_PI*a*b where a and b are the semi axis
// lengths area triangle = sqrt(s*(s-a)(s-b)(s-c)) where
// s=(a+b+c)/2
ld area() { // O(n)
  ld area = 0;
  forn(i, sz(pt)) area += pt[i] ^ pt[(i + 1) % sz(pt)];
  return abs(area) / ld(2);
}
// returns one pair of most distant points, convex only
pair<pto, pto> callipers() { // O(n), normalize first
  int n = sz(pt);
  if (n <= 2) return {pt[0], pt[1 % n]};
  pair<pto, pto> ret = {pt[0], pt[1]};
  T maxi = 0; int j = 1;
  forn(i, sz(pt)) {
    while(((pt[(i+1)%n]-pt[i])^(pt[(j+1)%n]-pt[j]))<-EPS)
      j = (j + 1) % sz(pt);
    if (pt[i].dist(pt[j]) > maxi + EPS)
      ret = {pt[i], pt[j]}, maxi = pt[i].dist(pt[j]);
  }
  return ret;
}
pto centroid(){//barycenter,mass center,needs float points
  int n = sz(pt); pto r(0, 0); ld t = 0;
  forn(i, n) {
    r = r+(pt[i] + pt[(i+1)%n]) * (pt[i] ^ pt[(i+1)%n]);
    t += pt[i] ^ pt[(i + 1) % n];
  }
  return r / t / 3;
 }
};
// Dynamic convex hull trick (based on poly struct)
vector<poly> w;
void add(pto q) { // add(q), O(log^2(n))
  vector<pto> p = {q};
  while (!w.empty() && sz(w.back().pt) < 2 * sz(p)) {
    for (pto v : w.back().pt) p.pb(v);
    w.pop_back();
  }
  w.pb(poly(CH(p)));//CH=convex hull, must delete collinears
}
T query(pto v) { // max(q*v:q in w), O(log^2(n))
  T r = -INF;
  for (auto& p : w) r = max(r, p.farthest(v) * v);
  return r;
}
```

3.12 Radial order

```
struct cmp {//sort around O in CCW direction starting from v
  pto o, v; // center point and starting vector
  cmp(pto no, pto nv) : o(no), v(nv) {}
  bool half(pto p) {
    assert(!(p.x == 0 && p.y == 0));//(0,0) not well defined
    return (v ^ p) < 0 || ((v ^ p) == 0 && (v * p) < 0);
  }
  bool operator()(pto& p1, pto& p2) {
    return mp(half(p1 - o), T(0)) <
           mp(half(p2 - o), ((p1 - o) ^ (p2 - o)));
  }
};
```

3.13 Segment

```
struct segm {
  pto s, e;
  segm(pto s_, pto e_) : s(s_), e(e_) {}
  pto closest(pto b) {
    pto bs = b - s, es = e - s;
    ld l = es * es;
    if (abs(l) <= EPS) return s;
    ld t = (bs * es) / l;
```

```
    if (t < 0.) return s;    // comment for lines
    else if (t > 1.) return e; // comment for lines
    return s + (es * t);
  }
  bool inside(pto b) {
    return abs(s.dist(b) + e.dist(b) - s.dist(e)) < EPS;
  }
  pto inter(segm b) { // if collinear, returns one point
    if ((*this).inside(b.s)) return b.s;
    if ((*this).inside(b.e)) return b.e;
    pto in = line(s, e).inter(line(b.s, b.e));
    if ((*this).inside(in) && b.inside(in)) return in;
    return pto(INF, INF);
  }
};
```

4 Graph

4.1 2-Sat

```
// 1. Create with n = number of variables (0-indexed)
// 2. Add restrictions (using ~X for negating variable X)
// 3. Call satisf() to check whether there is a solution
// 4. verdad[cmp[2*X]] for each variable X is a valid result
struct Sat2 {
  vector<vector<int>> G;
  // idx[i]=index assigned in the dfs
  // lw[i]=lowest index(closer from root) reachable from i
  // verdad[cmp[2*i]]=valor de la variable i
  int N, qidx, qcmp;
  vector<int> lw, idx, cmp, verdad; stack<int> q;
  Sat2(int n) : G(2 * n), N(n) {}
  void tjn(int v) {
    lw[v] = idx[v] = ++qidx; q.push(v), cmp[v] = -2;
    forall(it, G[v]) if (!idx[*it] || cmp[*it] == -2) {
      if (!idx[*it]) tjn(*it);
      lw[v] = min(lw[v], lw[*it]);
    }
    if (lw[v] == idx[v]) {
      int x;
      do { x=q.top(), q.pop(), cmp[x]=qcmp; } while (x!=v);
      verdad[qcmp] = (cmp[v ^ 1] < 0); qcmp++;
    }
  }
  bool satisf() { // O(N)
    idx = lw = verdad = vector<int>(2 * N, 0);
    cmp = vector<int>(2 * N, -1); qidx = qcmp = 0;
    forn(i, N) {
      if (!idx[2 * i]) tjn(2 * i);
      if (!idx[2 * i + 1]) tjn(2 * i + 1);
    }
    forn(i, N) if (cmp[2*i] == cmp[2*i+1]) return false;
    return true;
  }
  void addimpl(int a, int b) { // a -> b
    a = a >= 0 ? 2 * a : 2 * (~a) + 1; // avoid negatives
    b = b >= 0 ? 2 * b : 2 * (~b) + 1;
    G[a].pb(b), G[b ^ 1].pb(a ^ 1);
  }
  void addor(int a, int b){addimpl(~a, b);} // a|b = ~a->b
  void addeq(int a, int b){addimpl(a,b); addimpl(b,a);}//a=b
  void addxor(int a, int b){addeq(a, ~b);} // a xor b
  void force(int x, bool val) {
    if(val) addimpl(~x, x); else addimpl(x, ~x);
  }
  void atmost1(vector<int> v) { // At most 1 true in all v
    int auxid = N; N += sz(v); G.rsz(2 * N);
    forn(i, sz(v)) {
      addimpl(auxid, ~v[i]);
      if(i){addimpl(auxid,auxid-1); addimpl(v[i],auxid-1);}
      auxid++;
    }
    assert(auxid == N);
  }
};
```

4.2 Bellman Ford

```
// Can solve systems of "difference inequalities":
// 1. for each inequality x_i - x_j <= k add an edge j->i
// with weight k; 2. create an extra node Z and add an edge
// Z->i with weigth 0 for each variable x_i in the
// inequalities; 3. run(Z): if negcycle, no solution,
// otherwise "dist" is a solution
//
// Can transform a graph to get edges of positive weight
// (Jhonson algo): 1. Create an extra node Z and add edge
// Z->i with weight 0 for all nodes i; 2. Run bellman ford
// from Z; 3. For each original edge a->b (with weight w),
// change its weigt to be w+dist[a]-dist[b] (where dist is
// the result of step 2); 4. The shortest paths in the old
// and new graph are the same (their weight result may
// differ, but the paths are the same).
// This doesn't work well with neg cycles, but you can find
// them before step 3 and ignore all new weights that result
// in a neg value when executing step 3
struct BellmanFord {
  vector<vector<ii>> G; // pair = (weight, node)
  vector<ll> dist; int N;
  BellmanFord(int n) : G(n), N(n) {}
  void addEdge(int a, int b, ll w) { G[a].pb(mp(w, b)); }
  void run(int src) { // O(VE)
    dist = vector<ll>(N, INF); dist[src] = 0;
    forn(i, N - 1) forn(j, N) if (dist[j] != INF)
        forall(it, G[j]) dist[it->snd] =
            min(dist[it->snd], dist[j] + it->fst);
  }

  bool hasNegCycle() {
    forn(j, N) if (dist[j] != INF) forall(it, G[j])
      if (dist[it->snd] > dist[j] + it->fst) return true;
    // inside if: all points reachable from it->snd will
    // have -INF distance. But this is not enough to
    // identify which exact nodes belong to a neg cycle, nor
    // which can reach a neg cycle. To do so, you need to
    // run SCC and check whether each SCC hasNegCycle
    // independently. All nodes in a SCC that hasNegCycle
    // are part of a (not necessarily simple) neg cycle.
    return false;
  }
};
```

4.3 Biconnected

```
struct Bicon {
  vector<vector<int>> G;
  struct edge { int u, v, comp; bool bridge; };
  vector<edge> ve;
  void addEdge(int u, int v) {
    G[u].pb(sz(ve)), G[v].pb(sz(ve)); ve.pb({u,v,-1,false});
  }
  // d[i] = dfs id, b[i] = lowest id reachable from i
  // art[i]>0 iff i is an articulation point
  vector<int> d, b, art;
  int n, t, nbc, nart; // nbc = # of bicon comps
  Bicon(int nn) {
    n = nn; t = nbc = nart = 0; b = d = vector<int>(n, -1);
    art = vector<int>(n, 0); G = vector<vector<int>>(n);
    ve.clear();
  }
  stack<int> st;
  void dfs(int u, int pe) { // O(n + m)
    b[u] = d[u] = t++;
    forall(eid, G[u]) if (*eid != pe) {
      int v = ve[*eid].u ^ ve[*eid].v ^ u;
      if (d[v] == -1) {
```

```
      st.push(*eid); dfs(v, *eid);
      if (b[v] > d[u]) ve[*eid].bridge = true; // bridge
      if (b[v] >= d[u]) {                    // art
        if (art[u]++ == 0) nart++;
        int last; // start biconnected
        do { last=st.top(); st.pop(); ve[last].comp=nbc; }
        while (last != *eid);
        nbc++; // end biconnected
      }
      b[u] = min(b[u], b[v]);
    } else if (d[v] < d[u]) { // back edge
      st.push(*eid); b[u] = min(b[u], d[v]);
    }
  }
}
void run(){forn(i,n) if(d[i] == -1) art[i]--, dfs(i, -1);}
vector<set<int>> bctree; // block-cut tree, set to dedup
vector<int> artid; //art nodes to tree node (-1 for !arts)
void buildBlockCutTree() { // call run first!!
  // node id: [0, nbc) -> bc, [nbc, nbc+nart) -> art
  int ntree = nbc + nart, auxid = nbc;
  bctree = vector<set<int>>(ntree);
  artid = vector<int>(n, -1);
  forn(i, n) if (art[i] > 0) {
    forall(eid, G[i]) { // edges always bc <-> art
      // may want to add more data in bctree edges
      bctree[auxid].insert(ve[*eid].comp);
      bctree[ve[*eid].comp].insert(auxid);
    }
    artid[i] = auxid++;
  }
}
int getTreeIdForGraphNode(int u) {
  if (artid[u] != -1) return artid[u];
  if (!G[u].empty()) return ve[G[u][0]].comp;
  return -1; // for nodes with no neighbours in G
}
};
```

4.4 Centroid

```
struct Centroid {
  vector<vector<int>> g; vector<int> vp, vsz;
  vector<bool> taken;
  Centroid(int n) : g(n), vp(n), vsz(n), taken(n) {}
  void addEdge(int a, int b) { g[a].pb(b), g[b].pb(a); }
  void build() { centroid(0, -1, -1); } // O(nlogn)
  int dfs(int node, int p) {
    vsz[node] = 1;
    forall(it,g[node]) if(*it!=p && !taken[*it])
      vsz[node] += dfs(*it, node);
    return vsz[node];
  }
  void centroid(int node, int p, int cursz) {
    if (cursz == -1) cursz = dfs(node, -1);
    forall(it,g[node]) if(!taken[*it] && vsz[*it]>cursz/2){
      vsz[node] = 0, centroid(*it, p, cursz); return;
    }
    taken[node] = true, vp[node] = p;
    // do something using node as centroid
    forall(it,g[node]) if(!taken[*it])centroid(*it,node,-1);
  }
};
```

4.5 Dijkstra

```
struct Dijkstra { // WARNING: ii usually needs pair<ll, int>
  vector<vector<ii>> G; // ady list with pairs (weight, dst)
  vector<ll> dist;
  // vector<int> vp; // for path reconstruction (parent)
  int N;
  Dijkstra(int n) : G(n), N(n) {}
  void addEdge(int a, int b, ll w) { G[a].pb(mp(w, b)); }
  void run(int src) { // O(|E| log |V|)
    dist = vector<ll>(N, INF);
    // vp = vector<int>(N, -1);
    priority_queue<ii, vector<ii>, greater<ii>> Q;
    Q.push(make_pair(0, src)), dist[src] = 0;
    while (sz(Q)) {
      int node = Q.top().snd;
      ll d = Q.top().fst;
      Q.pop();
      if (d > dist[node]) continue;
      forall(it, G[node]) if (d + it->fst < dist[it->snd]) {
        dist[it->snd] = d + it->fst;
        // vp[it->snd] = node;
        Q.push(mp(dist[it->snd], it->snd));
      }
    }
  }
};
```

4.6 Euler Path

```
// Be careful, comments below assume that there are no nodes
// with degree 0. Euler [path/cycle] exists in a BIDIREC
// graph iff it is connected and have at most [2/0] odd
// degree nodes. Path starts from odd vertex when exists
// Euler [path/cycle] exists in a DIREC graph iff the graph
// is [connected when making edges bidirectional / a single
// SCC], and at most [1/0] node have indg - outdg = 1, at
// most [1/0] node have outdg - indg = 1, all the other
// nodes have indg = outdg. Start from node with outdg -
// indg = 1, when exists
struct edge { // Dir version (add commented code for undir)
  int y;
  // list<edge>::iterator rev;
  edge(int yy) : y(yy) {}
};
struct EulerPath {
  vector<list<edge>> g;
  EulerPath(int n) : g(n) {}
  void addEdge(int a, int b) {
    g[a].push_front(edge(b));
    // auto ia = g[a].begin(); g[b].push_front(edge(a));
    // auto ib = g[b].begin(); ia->rev=ib, ib->rev=ia;
  }
  vector<int> p;
  void go(int x) {
    while (sz(g[x])) {
      int y = g[x].front().y;
      // g[y].erase(g[x].front().rev);
      g[x].pop_front(); go(y);
    }
    p.push_back(x);
  }
  vector<int> getPath(int x){//get a path that starts from x
    // you must check that path exists from x before calling
    p.clear(), go(x); reverse(p.begin(), p.end()); return p;
  }
};
```

4.7 HLD

```
struct HLD {
  vector<int> w, p, dep; // weight,father,depth
  vector<vector<int>> g;
  HLD(int n) : w(n), p(n), dep(n), g(n), pos(n), head(n) {}
  void addEdge(int a, int b) { g[a].pb(b), g[b].pb(a); }
  void build(){p[0]=-1,dep[0]=0,dfs1(0),curpos=0,hld(0,-1);}
  void dfs1(int x) {
    w[x] = 1;
    for (int y : g[x]) if (y != p[x]) {
     p[y] = x, dep[y] = dep[x] + 1, dfs1(y); w[x] += w[y];
    }
```

```
  }
  int curpos; vector<int> pos, head;
  void hld(int x, int c) {
    if (c < 0) c = x;
    pos[x] = curpos++, head[x] = c;
    int mx = -1;
    for(int y:g[x])if(y!=p[x] && (mx<0 || w[mx]<w[y])) mx=y;
    if (mx >= 0) hld(mx, c);
    for (int y : g[x]) if (y != mx && y != p[x]) hld(y, -1);
  }
  // Here ST is segtree or other DS according to problem
  tipo query(int x, int y, ST& st) { // ST tipo
    tipo r = neutro;
    while (head[x] != head[y]) {
      if (dep[head[x]] > dep[head[y]]) swap(x, y);
      r = oper(r, st.get(pos[head[y]], pos[y]+1)); //ST oper
      y = p[head[y]];
    }
    if (dep[x] > dep[y]) swap(x, y);     // now x is lca
    r = oper(r, st.get(pos[x], pos[y] + 1)); // ST oper
    return r;
  }
};
// for point updates: st.set(pos[x],v) (x=node, v=new value)
// for lazy range upd: something similar to the query method
// for data on edge: -assign values of edges to "child" node
//           -change pos[x] to pos[x]+1 in query (line 31)
```

4.8 Kosaraju

```
struct Kosaraju {
  vector<vector<int>> G, gt; int N, cantcomp;
  vector<int> comp, used; stack<int> pila;
  Kosaraju(int n) : G(n), gt(n), N(n), comp(n) {}
  void addEdge(int a, int b) { G[a].pb(b), gt[b].pb(a); }
  void dfs1(int nodo) {
    used[nodo]=1; forall(it,G[nodo])if(!used[*it])dfs1(*it);
    pila.push(nodo);
  }
  void dfs2(int nodo) {
    used[nodo] = 2; comp[nodo] = cantcomp - 1;
    forall(it, gt[nodo]) if (used[*it] != 2) dfs2(*it);
  }
  void run() {
    cantcomp = 0; used = vector<int>(N, 0);
    forn(i, N) if (!used[i]) dfs1(i);
    while (!pila.empty()) {
      if(used[pila.top()]!=2){cantcomp++; dfs2(pila.top());}
      pila.pop();
    }
  }
};
```

4.9 Kruskal

```
struct Edge {
  int a, b, w;
};
bool operator<(const Edge& a, const Edge& b) {
    return a.w < b.w;
}
// Minimun Spanning Tree in O(E log E)
ll kruskal(vector<Edge> &E, int n) {
  ll cost = 0; sort(E.begin(), E.end());
  UnionFind uf(n);
  forall(it, E) if(!uf.join(it->a, it->b))
    cost += it->w;
  return cost;
}
```

4.10 LCA climb

```
#define lg(x) (31 - __builtin_clz(x)) //=floor(log2(x))
struct LCA { // Usage: 1) Create 2) Add edges 3) Call build
  int N, LOGN, ROOT;
  vector<int> L; // L[v] holds the level of v
  vector<vector<int>> vp, G; // vp[x][k] = 2^k ancestor of x
  LCA(int n, int root)
      : N(n), LOGN(lg(n) + 1), ROOT(root), L(n), G(n) {
    vp = vector<vector<int>>(n, vector<int>(LOGN, root));
  }
  void addEdge(int a, int b) { G[a].pb(b), G[b].pb(a); }
  void dfs(int node, int p, int lvl) {
    vp[node][0] = p, L[node] = lvl;
    forall(it, G[node]) if (*it != p) dfs(*it, node, lvl+1);
  }
  void build() {
    dfs(ROOT, ROOT, 0);
    forn(k, LOGN-1) forn(i, N) vp[i][k+1] = vp[vp[i][k]][k];
  }
  int climb(int a, int d) { // O(lgn)
    if (!d) return a;
    dforn(i, lg(L[a])+1) if(1<<i <= d) a=vp[a][i], d-=1<<i;
    return a;
  }
  int lca(int a, int b) { // O(lgn)
    if (L[a] < L[b]) swap(a, b);
    a = climb(a, L[a] - L[b]);
    if (a == b) return a;
    dforn(i, lg(L[a]) + 1) if (vp[a][i] != vp[b][i])
      a = vp[a][i], b = vp[b][i];
    return vp[a][0];
  }
  int dist(int a, int b){return L[a]+L[b] - 2*L[lca(a, b)];}
};
```

4.11 Tree reroot

```
struct Edge { int u, v; }; // maybe add more data
struct SubtreeData { // Define data for each subtree
  SubtreeData() {}  // just empty
  SubtreeData(int node) { /*implement this*/ }
  void merge(Edge* e, SubtreeData& s) { /*implement this*/ }
};
struct Reroot {
  int N; // # of nodes
  vector<SubtreeData> vresult, vs;//vresult[i], when root==i
  vector<Edge> ve;
  vector<vector<int>> g; // the tree as a bidirec graph
  Reroot(int n) : N(n), vresult(n), vs(n), ve(0), g(n) {}
  void addEdge(Edge e) {
    g[e.u].pb(sz(ve)); g[e.v].pb(sz(ve)); ve.pb(e);
  }
  void dfs1(int node, int p) {
    vs[node] = SubtreeData(node);
    forall(e, g[node]) {
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) continue;
      dfs1(nxt, node); vs[node].merge(&ve[*e], vs[nxt]);
    }
  }
  void dfs2(int node, int p, SubtreeData fromp) {
    vector<SubtreeData> vsuf(sz(g[node]) + 1);
    int pos = sz(g[node]);
    SubtreeData pref = vsuf[pos] = SubtreeData(node);
    vresult[node] = vs[node];
    dforall(e, g[node]) { // dforall = forall in reverse
      pos--; vsuf[pos] = vsuf[pos + 1];
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) {
        pref.merge(&ve[*e], fromp);
        vresult[node].merge(&ve[*e], fromp);
        continue;
      }
      vsuf[pos].merge(&ve[*e], vs[nxt]);
```

```
    }
    assert(pos == 0);
    forall(e, g[node]) {
      pos++; int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) continue;
      SubtreeData aux = pref;
      aux.merge(NULL, vsuf[pos]); dfs2(nxt, node, aux);
      pref.merge(&ve[*e], vs[nxt]);
    }
  }
  void run() { dfs1(0, 0); dfs2(0, 0, SubtreeData()); }
};
```

4.12 Virtual tree

```
struct VirtualTree {
  int n, curt; // n = #nodes full tree, curt used for dfs
  LCA* lca; vector<int> tin, tout;
  vector<vector<ii>> tree; // {node, dist}, p->child dir
  // imp[i] = true iff i was part of 'newv' last time that
  // updateVT was called (note that LCAs are not imp)
  vector<bool> imp;
  void dfs(int node, int p) {
    tin[node] = curt++;
    forall(it, lca->G[node]) if (*it != p) dfs(*it, node);
    tout[node] = curt++;
  }
  VirtualTree(LCA* l) { // must call l.build() before
    lca = l, n = sz(l->G), lca = l, curt = 0;
    tin.rsz(n), tout.rsz(n), tree.rsz(n), imp.rsz(n);
    dfs(l->ROOT, l->ROOT);
  }
  bool isAncestor(int a, int b) {
    return tin[a] < tin[b] && tout[a] > tout[b];
  }
  int VTroot = -1; // root of the current VT
  vector<int> v; // nodes of current VT (includes LCAs)
  void updateVT(vector<int>& newv) { // O(sz(newv)*log)
    assert(!newv.empty()); // this method assumes non-empty
    auto cmp = [this](int a,int b){return tin[a]<tin[b];};
    forn(i, sz(v)) tree[v[i]].clear(), imp[v[i]] = false;
    v = newv; sort(v.begin(), v.end(), cmp);
    set<int> s; forn(i,sz(v))s.insert(v[i]), imp[v[i]]=true;
    forn(i, sz(v) - 1) s.insert(lca->lca(v[i], v[i + 1]));
    v.clear(); forall(it, s) v.pb(*it);
    sort(v.begin(), v.end(), cmp);
    stack<int> st;
    forn(i, sz(v)) {
      while (!st.empty() && !isAncestor(st.top(), v[i]))
        st.pop();
      assert(i == 0 || !st.empty());
      if (!st.empty())
       tree[st.top()].pb(mp(v[i],lca->dist(st.top(),v[i])));
      st.push(v[i]);
    }
    VTroot = v[0];
  }
};
```

5 Math

5.1 CRT

```
// Chinese remainder theorem (special case): find z such
// that z % m1 = r1, z % m2 = r2. Here, z is unique modulo
// M = lcm(m1, m2).
// Return (z, M). On failure, M = -1.
//{xx,yy,d} son variables globales usadas en extendedEuclid
ii CRT(int m1,int r1,int m2,int r2){
  extendedEuclid(m1, m2);
  if (r1 % d != r2 % d) return make_pair(0, -1);
  return mp(sumMod(xx * r2 * m1, yy * r1 * m2, m1 * m2) / d,
            m1 * m2 / d);
}
// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.
// Note that the solution is unique modulo M = lcm_i (m[i]).
// Return (z, M). On failure, M = -1.
// Note that we dont require the a[i]'s to be relative prime
ii CRT(const vector<int>& m, const vector<int>& r) {
  ii ret = mp(r[0], m[0]);
  forr(i, 1, m.size()) {
    ret = CRT(ret.snd, ret.fst, m[i], r[i]);
    if (ret.snd == -1) break;
  }
  return ret;
}
```

5.2 Combinatorics

```
void cargarComb() { // O(MAXN^2)
  forn(i, MAXN) { // comb[i][k] = i!/(k!*(i-k)!)
    comb[0][i] = 0; comb[i][0] = comb[i][i] = 1;
    forr(k, 1, i)
      comb[i][k] = (comb[i-1][k-1] + comb[i-1][k]) % MOD;
  }
}
// returns comb(n,k)%p, needs comb[0..p][0..p] precalculated
ll lucas(ll n, ll k, int p) {
  ll aux = 1;
  while (n + k) {
    aux = (aux * comb[n % p][k % p]) % p; n /= p, k /= p;
  }
  return aux;
}
```

5.3 Discrete logarithm

```
// O(sqrt(m)*log(m))
// returns x such that a^x = b (mod m) or -1 if inexistent
ll discrete_log(ll a, ll b, ll m) {
  a %= m, b %= m;
  if (b == 1) return 0;
  int cnt = 0;
  ll tmp = 1;
  for (ll g = __gcd(a, m); g != 1; g = __gcd(a, m)) {
    if (b % g) return -1;
    m /= g, b /= g;
    tmp = tmp * a / g % m;
    ++cnt;
    if (b == tmp) return cnt;
  }
  map<ll, int> w;
  int s = (int)ceil(sqrt(m));
  ll base = b;
  forn(i, s) {
    w[base] = i;
    base = base * a % m;
  }
  base = expMod(a, s, m);
  ll key = tmp;
  forr(i, 1, s + 2) {
    key = base * key % m;
    if (w.count(key)) return i * s - w[key] + cnt;
  }
  return -1;
}
```

5.4 Extended euclid

```
// sea d=gcd(a,b); la ecuacion a * x + b * y = c tiene
// soluciones enteras si d|c. De forma general sera:
```

```
// x = x0 + (b/d)n   x0 = xx*c/d
// y = y0 - (a/d)n   y0 = yy*c/d
ll xx, yy, d;
void extendedEuclid(ll a, ll b) { // a * xx + b * yy = d
  if (!b) { xx = 1, yy = 0, d = a; return; }
  extendedEuclid(b, a % b);
  ll x1 = yy, y1 = xx - (a / b) * yy; xx = x1, yy = y1;
}
```

5.5 FFT

```
typedef __int128 T;
typedef double ld;
typedef vector<T> poly;
const T MAXN = (1 << 21); // MAXN must be power of 2,
// MOD-1 needs to be a multiple of MAXN,
// big mod and primitive root for NTT
const T MOD = 2305843009255636993LL, RT = 5;
// const T MOD = 998244353, RT = 3;

// NTT
struct CD {
  T x;
  CD(T x_) : x(x_) {}
  CD() {}
};
T mulmod(T a, T b) { return a * b % MOD; }
T addmod(T a, T b) {
  T r = a + b; if (r >= MOD) r -= MOD;
  return r;
}
T submod(T a, T b) {
  T r = a - b; if (r < 0) r += MOD;
  return r;
}
CD operator*(const CD& a, const CD& b) {
  return CD(mulmod(a.x, b.x));
}
CD operator+(const CD& a, const CD& b) {
  return CD(addmod(a.x, b.x));
}
CD operator-(const CD& a, const CD& b) {
  return CD(submod(a.x, b.x));
}
vector<T> rts(MAXN + 9, -1);
CD root(int n, bool inv) {
  T r = rts[n]<0 ? rts[n] = expMod(RT, (MOD-1)/n) : rts[n];
  return CD(inv ? expMod(r, MOD - 2) : r);
}

// FFT
// struct CD {
//   ld r, i;
//   CD(ld r_ = 0, ld i_ = 0) : r(r_), i(i_) {}
//   ld real() const { return r; }
//   void operator/=(const int c) { r /= c, i /= c; }
// };
// CD operator*(const CD& a, const CD& b) {
//   return CD(a.r*b.r - a.i*b.i, a.r*b.i + a.i*b.r);
// }
// CD operator+(const CD& a, const CD& b) {
//   return CD(a.r + b.r, a.i + b.i);
// }
// CD operator-(const CD& a, const CD& b) {
//   return CD(a.r - b.r, a.i - b.i);
// }
// const ld pi = acos(-1.0);

CD cp1[MAXN + 9], cp2[MAXN + 9];
int R[MAXN + 9];
void dft(CD* a, int n, bool inv) {
  forn(i, n) if (R[i] < i) swap(a[R[i]], a[i]);
  for (int m = 2; m <= n; m *= 2) {
    // ld z=2*pi/m*(inv?-1:1); // FFT
    // CD wi=CD(cos(z),sin(z)); // FFT
    CD wi = root(m, inv); // NTT
    for (int j = 0; j < n; j += m) {
      CD w(1);
      for (int k = j, k2 = j + m/2; k2 < j + m; k++, k2++) {
        CD u = a[k], v = a[k2] * w;
        a[k] = u + v; a[k2] = u - v; w = w * wi;
      }
    }
  }
  // if(inv) forn(i,n) a[i]/=n; // FFT
  if (inv) { // NTT
    CD z(expMod(n, MOD - 2));
    forn(i, n) a[i] = a[i] * z;
  }
}
poly multiply(poly& p1, poly& p2) {
  int n = sz(p1) + sz(p2) + 1, m = 1, cnt = 0;
  while (m <= n) m += m, cnt++;
  forn(i, m) {
    R[i] = 0;
    forn(j, cnt) R[i] = (R[i] << 1) | ((i >> j) & 1);
  }
  forn(i, m) cp1[i] = 0, cp2[i] = 0;
  forn(i, sz(p1)) cp1[i] = p1[i];
  forn(i, sz(p2)) cp2[i] = p2[i];
  dft(cp1, m, false); dft(cp2, m, false);
  forn(i, m) cp1[i] = cp1[i] * cp2[i];
  dft(cp1, m, true);
  poly res; n -= 2;
  // forn(i,n) res.pb((T) floor(cp1[i].real()+0.5)); // FFT
  forn(i, n) res.pb(cp1[i].x); // NTT
  return res;
}
```

5.6 Fraction

```
struct frac {
  int p, q;
  frac(int p = 0, int q = 1) : p(p), q(q) { norm(); }
  void norm() {
    int a = gcd(q, p);
    if (a) p /= a, q /= a; else q = 1;
    if (q < 0) q = -q, p = -p;
  }
  frac operator+(const frac& o) {
    int a = gcd(o.q, q);
    return frac(p * (o.q/a) + o.p * (q/a), q * (o.q/a));
  }
  frac operator-(const frac& o) {
    int a = gcd(o.q, q);
    return frac(p * (o.q/a) - o.p * (q/a), q * (o.q/a));
  }
  frac operator*(frac o) {
    int a = gcd(o.p, q), b = gcd(p, o.q);
    return frac((p/b) * (o.p/a), (q/a) * (o.q/b));
  }
  frac operator/(frac o) {
    int a = gcd(o.q, q), b = gcd(p, o.p);
    return frac((p/b) * (o.q/a), (q/a) * (o.p/b));
  }
  bool operator<(const frac& o) const{return p*o.q < o.p*q;}
  bool operator==(frac o) { return p == o.p && q == o.q; }
};
```

5.7 Gauss Jordan

```
// comments useful for MOD version
const double EPS = 1e-9; // remove for MOD version
const int INF = 2; // a value to indicate infinite solutions
// int instead of double for MOD
int gauss(vector<vector<double>> a, vector<double>& ans) {
```

```
  int n = sz(s), m = sz(a[0]) - 1;
  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    for (int i = row; i < n; ++i)
      // remove abs for MOD
      if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    // remove abs, and check == 0 for MOD
    if (abs(a[sel][col]) < EPS) continue;
    for (int i=col; i <= m; ++i) swap(a[sel][i], a[row][i]);
    where[col] = row;
    for (int i = 0; i < n; ++i) if (i != row) {
      double c = a[i][col] / a[row][col]; // inverse for MOD
      for(int j=col; j <= m; ++j) a[i][j]-=a[row][j]*c;//MOD
    }
    ++row;
  }
  ans.assign(m, 0);
  for (int i = 0; i < m; ++i)
    // use inverse for MOD
    if(where[i]!=-1) ans[i] = a[where[i]][m]/a[where[i]][i];
  for (int i = 0; i < n; ++i) {
    double sum = 0;
    for (int j=0; j < m; ++j) sum += ans[j] * a[i][j];//MOD
    // remove abs, and check != 0 for MOD
    if (abs(sum - a[i][m]) > EPS) return 0;
  }
  for (int i=0; i < m; ++i) if (where[i] == -1) return INF;
  return 1;
}

// Gauss Jordan with bitsets version, for MOD = 2
// finds lexicograhically minimal solution (0<1, False<True)
// for maximal change your solution model accordingly
int gauss(vector<bitset<N>> a, int n,int m, bitset<N>& ans){
  vector<int> where(m, -1);
  for (int col = m-1, row = 0; col >= 0 && row < n; --col) {
    for (int i = row; i < n; ++i) if (a[i][col]) {
      swap(a[i], a[row]); break;
    }
    if (!a[row][col]) continue;
    where[col] = row;
    for (int i = 0; i < n; ++i)
      if (i != row && a[i][col]) a[i] ^= a[row];
    ++row;
  }
  ans.reset();
  forn(i, m) if (where[i] != -1) {
    ans[i] = a[where[i]][m] & a[where[i]][i];
  }
  forn(i, n) if((ans & a[i]).count()%2 != a[i][m]) return 0;
  forn(i, m) if(where[i] == -1) return INF;
  return 1;
}
```

5.8 Karatsuba

```
void rec_kara(T* a, int one, T* b, int two, T* r) {
 if(min(one, two) <= 20) { // must be at least "<= 1"
  forn(i, one) forn(j, two) r[i+j] += a[i] * b[j];
  return;
 }
 const int x = min(one, two);
 if(one < two) rec_kara(a, x, b + x, two - x, r + x);
 if(two < one) rec_kara(a + x, one - x, b, x, r + x);
 const int n = (x + 1) / 2, right = x / 2;
 vector<T> tu(2 * n);
 rec_kara(a, n, b, n, tu.data());
 forn(i, 2*n-1) {
  r[i] += tu[i];
  r[i+n] -= tu[i];
  tu[i] = 0;
 }
 rec_kara(a + n, right, b + n, right, tu.data());
 forn(i, 2*right-1) r[i+n] -= tu[i], r[i+2*n] += tu[i];
 tu[n-1] = a[n-1]; tu[2*n-1] = b[n-1];
 forn(i,right) tu[i]=a[i]+a[i+n], tu[i+n]=b[i]+b[i+n];
 rec_kara(tu.data(), n, tu.data() + n, n, r + n);
}
vector<T> multiply(vector<T> a, vector<T> b) {
 if(a.empty() || b.empty()) return {};
 vector<T> r(a.size() + b.size() - 1);
 rec_kara(a.data(),a.size(),b.data(),b.size(),r.data());
 return r;
}
```

5.9 Matrix exponentiation

```
typedef ll tipo; // maybe use double or other
struct Mat {
  int N; // square matrix
  vector<vector<tipo>> m;
  Mat(int n) : N(n), m(n, vector<tipo>(n, 0)) {}
  vector<tipo>& operator[](int p) { return m[p]; }
  Mat operator*(Mat& b) { // O(N^3), multiplication
    assert(N == b.N); Mat res(N);
    forn(i,N) forn(j,N) forn(k,N)//remove MOD if not needed
      res[i][j] = (res[i][j] + m[i][k] * b[k][j]) % MOD;
    return res;
  }
  Mat operator^(int k) { // O(N^3 * logk), exponentiation
    Mat res(N), aux = *this; forn(i, N) res[i][i] = 1;
    while (k)
      if (k & 1) res = res * aux, k--;
      else aux = aux * aux, k /= 2;
    return res;
  }
};
```

5.10 Modular inverse

```
#define MAXMOD 15485867
ll inv[MAXMOD];   // inv[i]*i=1 mod MOD
void calc(int p) { // O(p)
  inv[1] = 1; forr(i,2,p) inv[i] = p - ((p/i) * inv[p%i])%p;
}
int inverso(int x) {                    // O(log MOD)
  return expMod(x, MOD - 2);            // if mod prime
  return expMod(x, eulerPhi(MOD) - 1); // if not prime
}
// fact[i] = i!%MOD and ifact[i] = 1/(i!)%MOD
// inv is modular inverse function
ll fact[MAXN], ifact[MAXN];
void build_facts() { // O(MAXN)
  fact[0] = 1; forr(i,1,MAXN) fact[i] = fact[i-1] * i % MOD;
  ifact[MAXN-1] = inverso(fact[MAXN-1]);
  dforn(i, MAXN-1) ifact[i] = ifact[i+1] * (i+1) % MOD;
  return;
}
```

5.11 Modular operations

```
// Only needed for MOD >= 2^31 (or 2^63)
// For 2^31 < MOD < 2^63 it's usually better to use __int128
// and normal multiplication (* operator) instead of mulMod
// returns (a*b) %c, and minimize overfloor
ll mulMod(ll a, ll b, ll m = MOD) { // O(log b)
  ll x = 0, y = a % m;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % m;
    y = (y * 2) % m;
    b /= 2;
  }
  return x % m;
```

```
}
ll expMod(ll b, ll e, ll m = MOD) { // O(log e)
  if (e < 0) return 0;
  ll ret = 1;
  while (e) {
    if (e & 1) ret = ret * b % m; // ret = mulMod(ret,b,m);
    b = b * b % m;             // b = mulMod(b,b,m);
    e >>= 1;
  }
  return ret;
}
```

5.12 Phollard-Rho

```
bool es_primo_prob(ll n, int a) {
  if (n == a) return true;
  ll s = 0, d = n - 1;
  while (d % 2 == 0) s++, d /= 2;
  ll x = expMod(a, d, n);
  if ((x == 1) || (x + 1 == n)) return true;
  forn(i, s - 1) {
    x = (x * x) % n; // mulMod(x, x, n);
    if (x == 1) return false;
    if (x + 1 == n) return true;
  }
  return false;
}
bool rabin(ll n) { // devuelve true si n es primo
  if (n == 1) return false;
  const int ar[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  forn(j, 9) if (!es_primo_prob(n, ar[j])) return false;
  return true;
}
ll rho(ll n) {
  if ((n & 1) == 0) return 2;
  ll x = 2, y = 2, d = 1;
  ll c = rand() % n + 1; // maybe use rng
  while (d == 1) {
    x = (mulMod(x, x, n) + c) % n; // try to avoid mulmod
    forn(_,2) y = (mulMod(y, y, n) + c) % n;
    if (x-y >= 0) d = gcd(n, x-y); else d = gcd(n, y-x);
  }
  return d == n ? rho(n) : d;
}
void factRho(ll n, map<ll, ll>& f) { // O ((n ^ 1/4) * logn)
  if (n == 1) return;
  if (rabin(n)) { f[n]++; return; }
  ll factor = rho(n); factRho(factor,f);factRho(n/factor,f);
}
```

5.13 Primes

```
#define MAXP 100000 // no necesariamente primo
int criba[MAXP + 1];
void crearCriba() {
  int w[] = {4, 2, 4, 2, 4, 6, 2, 6};
  for (int p = 25; p <= MAXP; p += 10) criba[p] = 5;
  for (int p = 9; p <= MAXP; p += 6) criba[p] = 3;
  for (int p = 4; p <= MAXP; p += 2) criba[p] = 2;
  for (int p = 7, cur = 0; p * p <= MAXP; p += w[cur++ & 7])
    if (!criba[p])
      for (int j = p * p; j <= MAXP; j += (p << 1))
        if (!criba[j]) criba[j] = p;
}
vector<int> primos;
void buscarPrimos() {
  crearCriba();
  forr(i, 2, MAXP + 1) if (!criba[i]) primos.push_back(i);
}
// facts up to MAXP^2, call buscarPrimos first
void fact(ll n, map<ll, ll>& f) { // O (# primos)
  forall(p, primos) {
    while (!(n % *p)) {
      f[*p]++; // divisor found
      n /= *p;
    }
  }
  if (n > 1) f[n]++;
}
// facts up to MAXP, call crearCriba first
void fact2(ll n, map<ll, ll>& f) { // O (lg n)
  while (criba[n]) {
    f[criba[n]]++;
    n /= criba[n];
  }
  if (n > 1) f[n]++;
}
// Use: divisores(fac, divs, fac.begin()); NOT ORDERED
void divisores(map<ll, ll>& f, vector<ll>& divs, map<ll, ll
     >::iterator it,
               ll n = 1) {
  if (it == f.begin()) divs.clear();
  if (it == f.end()) {
    divs.pb(n);
    return;
  }
  ll p = it->fst, k = it->snd;
  ++it;
  forn(_, k + 1) divisores(f, divs, it, n), n *= p;
}
ll sumDivs(map<ll, ll>& f) {
  ll ret = 1;
  forall(it, f) {
    ll pot = 1, aux = 0;
    forn(i, it->snd + 1) aux += pot, pot *= it->fst;
    ret *= aux;
  }
  return ret;
}
ll eulerPhi(ll n) { // con criba: O(lg n)
  map<ll, ll> f;
  fact(n, f);
  ll ret = n;
  forall(it, f) ret -= ret / it->first;
  return ret;
}
ll eulerPhi2(ll n) { // O (sqrt n)
  ll r = n;
  forr(i, 2, n + 1) {
    if ((ll)i * i > n) break;
    if (n % i == 0) {
      while (n % i == 0) n /= i;
      r -= r / i;
    }
  }
  if (n != 1) r -= r / n;
  return r;
}
```

5.14 Simplex

```
typedef double tipo;
typedef vector<tipo> vt;
// maximize c^T x s.t. Ax<=b, x>=0,
// returns pair (max val, solution vector)
pair<tipo, vt> simplex(vector<vt> A, vt b, vt c) {
  int n = sz(b), m = sz(c);
  tipo z = 0.;
  vector<int> X(m), Y(n);
  forn(i, m) X[i] = i;
  forn(i, n) Y[i] = i + m;
  auto pivot = [&](int x, int y) {
    swap(X[y], Y[x]);
    b[x] /= A[x][y];
    forn(i, m) if (i != y) A[x][i] /= A[x][y];
    A[x][y] = 1 / A[x][y];
```

```
    forn(i, n) if (i != x && abs(A[i][y]) > EPS) {
      b[i] -= A[i][y] * b[x];
      forn(j, m) if (j != y) A[i][j] -= A[i][y] * A[x][j];
      A[i][y] *= -A[x][y];
    }
    z += c[y] * b[x];
    forn(i, m) if (i != y) c[i] -= c[y] * A[x][i];
    c[y] *= -A[x][y];
  };
  while (1) {
    int x = -1, y = -1;
    tipo mn = -EPS;
    forn(i, n) if (b[i] < mn) mn = b[i], x = i;
    if (x < 0) break;
    forn(i, m) if (A[x][i] < -EPS) { y = i; break; }
    assert(y >= 0); // no solution to Ax<=b
    pivot(x, y);
  }
  while (1) {
    tipo mx = EPS;
    int x = -1, y = -1;
    forn(i, m) if (c[i] > mx) mx = c[i], y = i;
    if (y < 0) break;
    tipo mn = 1e200;
    forn(i, n) if (A[i][y] > EPS && b[i] / A[i][y] < mn) {
      mn = b[i] / A[i][y], x = i;
    }
    assert(x >= 0); // c^T x is unbounded
    pivot(x, y);
  }
  vt r(m); forn(i, n) if (Y[i] < m) r[Y[i]] = b[i];
  return {z, r};
}
```

5.15 Simpson

```
// polar coords: x=r*cos(theta), y=r*sin(theta), f=(r*r)/2
T simpson(std::function<T(T)> f,T a,T b,int n=10000){//O(n)
  T area = 0, h = (b - a) / T(n), fa = f(a), fb;
  forn(i, n) {
    fb = f(a + h * T(i + 1));
    area += fa + T(4) * f(a + h * T(i + 0.5)) + fb;
    fa = fb;
  }
  return area * h / T(6.); // T usually double
}
```

6 Strings

6.1 Aho Corasick

```
struct Node {
  map<char, int> next, go; int p, link, leafLink;
  char pch; vector<int> leaf;
  Node(int pp, char c):p(pp),link(-1),leafLink(-1),pch(c) {}
};
struct AhoCorasick {
  vector<Node> t = {Node(-1, -1)};
  void add_string(string s, int id) {
    int v = 0;
    for (char c : s) {
      if (!t[v].next.count(c)) {
     t[v].next[c] = sz(t); t.pb(Node(v, c));
    }
      v = t[v].next[c];
    }
    t[v].leaf.pb(id);
  }
  int go(int v, char c) {
    if (!t[v].go.count(c)) {
      if (t[v].next.count(c)) t[v].go[c] = t[v].next[c];
      else t[v].go[c] = v == 0 ? 0 : go(get_link(v), c);
    }
    return t[v].go[c];
  }
  int get_link(int v) { // suffix link
    if (t[v].link < 0) {
      if (!v || !t[v].p) t[v].link = 0;
      else t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
  }
  int get_leaf_link(int v) {//like suffix link, but to root
    if (t[v].leafLink < 0) {//or node with !empty leaf list
      if (!v || !t[v].p) t[v].leafLink = 0;
      else if (!t[get_link(v)].leaf.empty())
        t[v].leafLink = t[v].link;
      else t[v].leafLink = get_leaf_link(t[v].link);
    }
    return t[v].leafLink;
  }
};
```

6.2 Hash

```
const int P = 1777771, MOD[2] = {999727999, 1070777777};
const int PI[2]={325255434,10018302};//PI[i] = P^-1 % MOD[i]
struct Hash {
  vector<int> h[2], pi[2];
  vector<ll> vp[2]; // Only for getChanged
  Hash(string& s) {
    forn(k, 2)
      h[k].rsz(sz(s)+1), pi[k].rsz(sz(s)+1),
      vp[k].rsz(sz(s)+1);
    forn(k, 2) {
      ll p = 1; h[k][0] = 0; vp[k][0] = pi[k][0] = 1;
      forr(i, 1, sz(s) + 1) {
        h[k][i] = (h[k][i - 1] + p * s[i - 1]) % MOD[k];
        pi[k][i] = (1LL * pi[k][i - 1] * PI[k]) % MOD[k];
        vp[k][i] = p = (p * P) % MOD[k];
      }
    }
  }
  ll get(int s, int e) {//get hash value of substring [s, e)
    ll H[2];
    forn(i, 2) {
      H[i] = (h[i][e] - h[i][s] + MOD[i]) % MOD[i];
      H[i] = (1LL * H[i] * pi[i][s]) % MOD[i];
    }
    return (H[0] << 32) | H[1];
  }
  // get hash value of [s, e) if origVal in pos is changed
  // to val. For multiple changes, do what is done in the
  // for loop for each change
  ll getChanged(int s,int e,int pos,int val,int origVal) {
    ll hv = get(s, e), hh[2];
    hh[1] = hv & ((1LL << 32) - 1); hh[0] = hv >> 32;
    forn(i, 2)
      hh[i]=(hh[i]+vp[i][pos]*(val-origVal+MOD[i]))%MOD[i];
    return (hh[0] << 32) | hh[1];
  }
};
```

6.3 KMP

```
// b[i] = longest border of t[0,i) = length of the longest
// prefix of the substring P[0..i-1) that is also suffix of
// the substring P[0..i). For "AABAACAABAA":
// b[i] = {-1, 0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}
vector<int> kmppre(string& P) {
  vector<int> b(sz(P) + 1); b[0] = -1; int j = -1;
  forn(i, sz(P)) {
    while (j >= 0 && P[i] != P[j]) j = b[j];
    b[i + 1] = ++j;
```

```
  }
  return b;
}
void kmp(string& T, string& P) {//Text, Pattern, O(|T|+|P|)
  int j = 0; vector<int> b = kmppre(P);
  forn(i, sz(T)) {
    while (j >= 0 && T[i] != P[j]) j = b[j];
    if (++j == sz(P))
      j = b[j]; // Match at i-j+1, use it before assignment
  }
}
```

6.4 LCP

```
// LCP(sa[i], sa[j]) = min(lcp[i+1], lcp[i+2], ..., lcp[j])
// example "banana", sa = {5,3,1,0,4,2}, lcp = {0,1,3,0,0,2}
// Num of dif substrings: (n*n+n)/2 - (sum over lcp array)
// Build suffix array (sa) before calling
vector<int> computeLCP(string& s, vector<int>& sa) {
  int n = s.size(), L = 0;
  vector<int> lcp(n), plcp(n), phi(n);
  phi[sa[0]] = -1;
  forr(i, 1, n) phi[sa[i]] = sa[i - 1];
  forn(i, n) {
    if (phi[i] < 0) {
      plcp[i] = 0;
      continue;
    }
    while (s[i + L] == s[phi[i] + L]) L++;
    plcp[i] = L;
    L = max(L - 1, 0);
  }
  forn(i, n) lcp[i] = plcp[sa[i]];
  return lcp; // lcp[i]=LCP(sa[i-1],sa[i])
}
```

6.5 Manacher

```
int d1[MAXN]; // d1[i] = max odd palindrome centered on i
int d2[MAXN]; // d2[i] = max even palindrome centered on i
// s  aabbaacaabbaa
// d1 1111117111111
// d2 0103010010301
void manacher(string& s) { // O(|S|)
  int l = 0, r = -1, n = s.size();
  forn(i, n) { // build d1
    int k = i > r ? 1 : min(d1[l+r-i], r-i);
    while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) k++;
    d1[i] = k--;
    if (i+k > r) l = i-k, r = i+k;
  }
  l = 0, r = -1;
  forn(i, n) { // build d2
    int k = (i > r ? 0 : min(d2[l+r-i+1], r-i+1)) + 1;
    while (i+k <= n && i-k >= 0 && s[i+k-1] == s[i-k]) k++;
    d2[i] = --k;
    if (i+k-1 > r) l = i-k, r = i+k-1;
  }
}
```

6.6 Suffix Tree

```
struct SuffixTree {
  vector<char> s;
  vector<map<int, int>> to; // fst char of edge -> node
  // s[fpos[i], fpos[i]+len[i]): susbtr on edge from i's
  // father to i.
  // link[i] goes to the node that corresponds to the substr
  // that result after "removing" the first char of the
  // substr that i represents. Not defined for leaf nodes.
  vector<int> len, fpos, link;
  SuffixTree() { make_node(0, INF); }// maybe reserve memory
  int node = 0, pos = 0, n = 0;
  int make_node(int p, int l) {
    fpos.pb(p), len.pb(l), to.pb({}), link.pb(0);
    return sz(to) - 1;
  }
  void go_edge() {
    while (pos > len[to[node][s[n - pos]]]) {
      node = to[node][s[n - pos]];
      pos -= len[node];
    }
  }
  void add(char c) {
    int last = 0; s.pb(c), n++, pos++;
    while (pos > 0) {
      go_edge(); int edge = s[n - pos];
      int& v = to[node][edge]; int t = s[fpos[v] + pos-1];
      if (v == 0) {
        v=make_node(n-pos, INF); link[last]=node; last=0;
      } else if (t == c) { link[last] = node; return; }
      else {
        int u = make_node(fpos[v], pos - 1);
        to[u][c] = make_node(n - 1, INF);
        to[u][t] = v; fpos[v] += pos-1, len[v] -= pos-1;
        v = u, link[last] = u, last = u;
      }
      if (node == 0) pos--;
      else node = link[node];
    }
  }
  void finishedAdding() {
    forn(i,sz(len)) if(len[i]+fpos[i]>n) len[i] = n-fpos[i];
  }
  // Map each suffix with it corresponding leaf node
  // vleaf[i] = node id of leaf of suffix s[i..n)
  // The last character of the string must be unique. Use
  // 'buildLeaf' not 'dfs' directly. Also 'finishedAdding'
  // must be called before calling 'buildLeaf'. When this is
  // needed, usually binary lifting (vp) and depths are also
  // needed. Usually you also need to compute extra
  // information in the dfs
  vector<int> vleaf, vdepth; vector<vector<int>> vp;
  void dfs(int cur, int p, int curlen) {
    if (cur > 0) curlen += len[cur];
    vdepth[cur] = curlen; vp[cur][0] = p;
    if (to[cur].empty()) {
      assert(0 < curlen && curlen <= n);
      assert(vleaf[n - curlen] == -1);
      vleaf[n - curlen] = cur;
      // here maybe compute some extra info
    } else forall(it, to[cur]) {
        dfs(it->snd, cur, curlen);
        // here maybe compute some extra info
      }
  }
  void buildLeaf() {
    vdepth.rsz(sz(to), 0);             // tree size
    vleaf.rsz(n, -1);                  // string size
    vp.rsz(sz(to), vector<int>(MAXLOG)); // tree size * log
    dfs(0, 0, 0);
    forr(k, 1, MAXLOG) forn(i, sz(to))
      vp[i][k] = vp[vp[i][k - 1]][k - 1];
    forn(i, n) assert(vleaf[i] != -1);
  }
};
```

6.7 Suffix array slow

```
pair<int, int> sf[MAXN];
bool sacomp(int lhs, int rhs) { return sf[lhs] < sf[rhs]; }
vector<int> constructSA(string& s) { // O(n log^2(n))
  int n = s.size();
  vector<int> sa(n), r(n);
  // r[i]: equivalence class of s[i..i+m)
```

```
  forn(i, n) r[i] = s[i];
  for (int m = 1; m < n; m *= 2) {
    // sf[i] = {r[i], r[i+m]},
    // used to sort for next equivalence classes
    forn(i, n) sa[i] = i, sf[i] = {r[i], i+m<n?r[i+m]:-1};
    stable_sort(sa.begin(), sa.end(), sacomp);// O(n log(n))
    r[sa[0]] = 0;
    // if sf[sa[i]] == sf[sa[i-1]] then same equiv class
    forr(i,1,n)r[sa[i]]=sf[sa[i]]!=sf[sa[i-1]]?i:r[sa[i-1]];
  }
  return sa;
}
```

6.8 Suffix array

```
#define RB(x) (x < n ? r[x] : 0)
void csort(vector<int>& sa, vector<int>& r, int k) {
  int n = sa.size();
  vector<int> f(max(255, n), 0), t(n);
  forn(i, n) f[RB(i + k)]++;
  int sum = 0;
  forn(i, max(255, n)) f[i] = (sum += f[i]) - f[i];
  forn(i, n) t[f[RB(sa[i] + k)]++] = sa[i];
  sa = t;
}
vector<int> constructSA(string& s) { // O(n logn)
  int n = s.size(), rank;
  vector<int> sa(n), r(n), t(n);
  // r[i]: equivalence class of s[i..i+k)
  forn(i, n) sa[i] = i, r[i] = s[i];
  for (int k = 1; k < n; k *= 2) {
    csort(sa, r, k);
    csort(sa, r, 0); // counting sort, O(n)
    // t : equiv classes array for next size
    t[sa[0]] = rank = 0;
    forr(i, 1, n) {
      // check if sa[i] and sa[i-1] are in same equiv class
      if (r[sa[i]] != r[sa[i - 1]] ||
          RB(sa[i] + k) != RB(sa[i - 1] + k))
        rank++;
      t[sa[i]] = rank;
    }
    r = t;
    if (r[sa[n - 1]] == n - 1) break;
  }
  return sa;
}
```

6.9 Trie

```
struct Trie {
  map<char, Trie> m; // Trie* when using persistence
  // For persistent trie only. Call "clone" probably from
  // "add" and/or other methods, to implement persistence.
  void clone(int pos) {
    Trie* prev = NULL;
    if (m.count(pos)) prev = m[pos];
    m[pos] = new Trie();
    if (prev != NULL) {
      m[pos]->m = prev->m;
      // copy other relevant data
    }
  }
  void add(const string& s, int p = 0) {
    if (s[p]) m[s[p]].add(s, p + 1);
  }
  void dfs() {
    // Do stuff
    forall(it, m) it->second.dfs();
  }
};
```

6.10 Z Function

```
// z[i] = length of longest substring starting from S[i]
// that is prefix of S. z[i] = max k: S[0,k) == S[i,i+k)
vector<int> zFunction(string& s) {
  int l = 0, r = 0, n = sz(s); vector<int> z(n, 0);
  forr(i, 1, n) {
    if (i <= r) z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
void match(string& T, string& P) {//Text, Pattern O(|T|+|P|)
  string s = P + '$' + T; vector<int> z = zFunction(s);
  forr(i,sz(P)+1,sz(s)) if(z[i]==sz(P));//match at i-sz(P)-1
}
```

7 Utils and other

7.1 C++ utils

```
// 1- (mt19937_64 for 64-bits version)
mt19937 rng(
    chrono::steady_clock::now().time_since_epoch().count());
shuffle(v.begin(), v.end(), rng); // vector random shuffle
// 2- Pragma
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
// 3- Custom comparator for set/map
struct comp {
  bool operator()(const double& a, const double& b) const {
    return a + EPS < b;
  }
};
set<double, comp> w; // or map<double,int,comp>
// 4- Iterate over non empty subsets of bitmask
for (int s = m; s; s = (s - 1) & m) // Decreasing order
for (int s = 0; s = s - m & m;)  // Increasing order
// 5- Other bits operations
int __builtin_popcount(unsigned int x) // # of bits on in x
int __builtin_popcountll(unsigned long long x) // ll version
int __builtin_ctz(unsigned int x) //# of trailing 0 (x != 0)
int __builtin_clz(unsigned int x) // # of leading 0 (x != 0)
v = (x & (-x)) // Find the least significant bit that is on
// 6- Input
inline void Scanf(int& a) { // only positive ints
  char c = 0;
  while (c < 33) c = getc(stdin);
  a = 0;
  while (c > 33) a = a * 10 + c - '0', c = getc(stdin);
}
```

7.2 Compile Commands

g++ -std=c++20 file -o filename
Para Geany:
compile: g++ -DANARAP -std=c++20 -g -O2 -Wconversion -Wshadow -Wall -Wextra -c "%f"
build: g++ -DANARAP -std=c++20 -g -O2 -Wconversion -Wshadow -Wall -Wextra -o "%e" "%f"

7.3 DQ dp

```
vector<vector<int>> dp; //maybe replace dim of K with only 2
int n;
// d&q DP: go down the range [l,r) like merge sort, but also
// making sure to iterate over [from,to) in each step, and
```

```
// spliting the [from,to) in 2 parts when goind down:
// [from, best] and [best, to)
void solve(int l, int r, int k, int from, int to) {
 if(l >= r) return;
 int cur = (l+r)/2;
 int bestpos = cur-1;
 int best = INF; // assumes we want to minimize cost
 forr(i,from,min(cur, to)) {
  // cost function that usually depends on dp[i][k-1]
  int c = fcost(i, k);
  if(c < best) best = c, bestpos = i;
 }
 dp[cur][k] = best;
 solve(l, cur, k, from, bestpos+1);
 solve(cur+1, r, k, bestpos, to);
}
```

7.4 Mo's

```
// Commented code should be used if updates are needed
int n, sq, nq; // array size, sqrt(array size), #queries
struct Qu {   //[l, r)
  int l, r, id;
  // int upds; // # of updates before this query
};
Qu qs[MAXN];
ll ans[MAXN]; // ans[i] = answer to ith query
// struct Upd{
//   int p, v, prev; // pos, new_val, prev_val
// };
// Upd vupd[MAXN];

// Without updates
bool qcomp(const Qu& a, const Qu& b) {
  if (a.l / sq != b.l / sq) return a.l < b.l;
  return (a.l / sq) & 1 ? a.r < b.r : a.r > b.r;
}
// With updates
// bool qcomp(const Qu &a, const Qu &b){
//   if(a.l/sq != b.l/sq) return a.l<b.l;
//   if(a.r/sq != b.r/sq) return a.r<b.r;
//   return a.upds < b.upds;
// }

// Without updates: O(n^2/sq + q*sq)
// with sq = sqrt(n): O(n*sqrt(n) + q*sqrt(n))
// with sq = n/sqrt(q): O(n*sqrt(q))
//
// With updates: O(sq*q + q*n^2/sq^2)
// with sq = n^(2/3): O(q*n^(2/3))
// with sq = (2*n^2)^(1/3) may improve a bit
void mos() {
  forn(i, nq) qs[i].id = i;
  sq = sqrt(n) + .5; // without updates
  // sq=pow(n, 2/3.0)+.5; // with updates
  sort(qs, qs + nq, qcomp);
  int l = 0, r = 0;
  init();
  forn(i, nq) {
    Qu q = qs[i];
    while (l > q.l) add(--l);
    while (r < q.r) add(r++);
    while (l < q.l) remove(l++);
    while (r > q.r) remove(--r);
    // while(upds<q.upds){
    //   if(vupd[upds].p >= l && vupd[upds].p < r)
    //     remove(vupd[upds].p);
    //   v[vupd[upds].p] = vupd[upds].v; // do update
    //   if(vupd[upds].p >= l && vupd[upds].p < r)
    //     add(vupd[upds].p);
    //   upds++;
    // }
    // while(upds>q.upds){
    //   upds--;
    //   if(vupd[upds].p >= l && vupd[upds].p < r)
    //     remove(vupd[upds].p);
    //   v[vupd[upds].p] = vupd[upds].prev; // undo update
    //   if(vupd[upds].p >= l && vupd[upds].p < r)
    //     add(vupd[upds].p);
    // }
    ans[q.id] = get_ans();
  }
}
```

7.5 Python example

```
import sys, math
input = sys.stdin.readline
############ ---- Input Functions ---- ############
def inp():
    return(int(input()))
def inlt():
    return(list(map(int,input().split())))
def insr():
    s = input()
    return(list(s[:len(s) - 1]))
def invr():
    return(map(float,input().split()))
n, k = inlt() # read two numbers in a line
```

7.6 Template

```
#include <bits/stdc++.h>
#define forr(i, a, b) for (int i = (a); i < (b); i++)
#define forn(i, n) forr(i, 0, n)
#define dforn(i, n) for (int i = (n) - 1; i >= 0; i--)
#define forall(it,v) for(auto it=v.begin();it!=v.end();it++)
#define sz(c) ((int)c.size())
#define rsz resize
#define pb push_back
#define mp make_pair
#define lb lower_bound
#define ub upper_bound
#define fst first
#define snd second
using namespace std;
typedef long long ll;
typedef pair<int, int> ii;

int main() {
#ifdef ANARAP
  freopen("input.in", "r", stdin);
#endif
  ios::sync_with_stdio(false);
  cin.tie(NULL); cout.tie(NULL);
  return 0;
}
```

7.7 Theory

Derangements: Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor \frac{n!}{e} \right\rceil$$

Burnside's lemma: Given a group G of symmetries and a set X, the number of elements of X *up to symmetry* equals

$$\frac{1}{|G|}\sum_{g\in G}|X^g|,$$

where X^g are the elements fixed by g $(g.x = x)$. If $f(n)$ counts "configurations" (of some sort) of length n, we can ignore rotational symmetry using $G = Z_n$ to get

$$g(n) = \frac{1}{n}\sum_{k=0}^{n-1} f(\gcd(n,k)) = \frac{1}{n}\sum_{k|n} f(k)\phi(n/k).$$

Labeled unrooted trees: # on n vertices: n^{n-2}
on k existing trees of size n_i: $n_1 n_2 \cdots n_k n^{k-2}$
with degrees d_i: $(n-2)!/((d_1-1)!\cdots(d_n-1)!)$
Catalan numbers:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2}C_n,\ C_{n+1} = \sum C_i C_{n-i}$$

$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- binary search trees with n vertices.
- binary trees with n nodes is $C_n * n!$.
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Stars and bars: Count number of ways to partition a set of n unlabeled objects into k (possibly empty) labeled subsets:

$$\binom{n+k-1}{n}$$

Stirling numbers: Count number of ways to partition a set of n labeled objects into k nonempty unlabeled subsets:

$$S_{n,k} = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\binom{k}{i}i^n = \sum_{i=0}^{k}\frac{(-1)^{k-i}i^n}{(k-i)!i!}$$

Bell numbers: Count number of partitions of a set with n members:

$$B_n = \sum_{k=0}^{n} S_{n,k}$$

Number of Spanning Trees: Create an $N \times N$ matrix `mat`, and for each edge $a \to b \in G$, do `mat[a][b]--`, `mat[b][b]++` (and `mat[b][a]--`, `mat[a][a]++` if G is undirected). Remove the ith row and column and take the determinant; this yields the number of directed spanning trees rooted at i (if G is undirected, remove any row/column).
Erdős–Gallai theorem: A simple graph with node degrees $d_1 \geq \cdots \geq d_n$ exists iff $d_1 + \cdots + d_n$ is even and for every $k = 1 \ldots n$,

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n}\min(d_i, k).$$

Equations:

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \Rightarrow \begin{aligned} x &= \frac{ed - bf}{ad - bc} \\ y &= \frac{af - ec}{ad - bc} \end{aligned}$$

Triangles: Side lengths: a, b, c
Semiperimeter: $p = \dfrac{a+b+c}{2}$
Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$
Circumradius: $R = \dfrac{abc}{4A}$
Inradius: $r = \dfrac{A}{p}$
Length of median (divides triangle into two equal-area triangles): $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Spherical coordinates:

$$\begin{aligned} x &= r\sin\theta\cos\phi & r &= \sqrt{x^2+y^2+z^2} \\ y &= r\sin\theta\sin\phi & \theta &= \mathrm{acos}(z/\sqrt{x^2+y^2+z^2}) \\ z &= r\cos\theta & \phi &= \mathrm{atan2}(y, x) \end{aligned}$$

Sums:

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c-1}, c \neq 1$$

$$\begin{aligned} 1 + 2 + 3 + \cdots + n &= \frac{n(n+1)}{2} \\ 1^2 + 2^2 + 3^2 + \cdots + n^2 &= \frac{n(2n+1)(n+1)}{6} \\ 1^3 + 2^3 + 3^3 + \cdots + n^3 &= \frac{n^2(n+1)^2}{4} \\ 1^4 + 2^4 + 3^4 + \cdots + n^4 &= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \end{aligned}$$

Series:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty) \\ \ln(1+x) &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1) \\ \sqrt{1+x} &= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1) \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty) \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty) \end{aligned}$$

Lagrange interpolation: $n+1$ points determine the following polynomial of degree n:

$$f(x) = \sum_{i=1}^{n} y_i \prod_{i \neq j}\frac{x - x_j}{x_i - x_j}$$

Note: denominator can be get efficently if x coordinates are all from 1 to $n+1$ using suffix and prefix arrays.

Expectation is linear:

$$E(aX + bY) = aE(X) + bE(Y)$$

Pick's theorem: $A = I + \frac{B}{2} - 1$
Konig's Theorem: In a bipartite graph, max matching = min vertex cover (cover edges using nodes). Also, min edge cover (cover nodes using edges) = max independent set = N - min vertex cover = N - max matching
Dilworth's Theorem: An antichain in a partially ordered set is a set of elements no two of which are comparable to each other, and a chain is a set of elements every two of which are comparable. A chain decomposition is a partition of the elements of the order into disjoint chains. Dilworth's theorem states that for any partially ordered set, the sizes of the max antichain and of the min chain decomposition are equal. Equivalent to Konig's theorem on the bipartite graph (U, V, E) where $U = V = S$ and (u, v) is an edge when $u < v$. Those vertices outside the min vertex cover in both U and V form a max antichain.