Universiteit Utrecht

Team Code Reference

Curiously Recurring

ACM-ICPC World Finals

May 19, 2016

1 Templates

1.1 Vimrc

```
syntax on noet wrap lbr nu is cin ai
ts=4 sts=4 sw=4  mouse=nvc cb=unnamed bs=indent,eol,start cino=:0,l1,g0,(0
```

1.2 C++ Template

```cpp
//#define _GLIBCXX_DEBUG
#include <bits/stdc++.h>
//iostream string sstream vector list set map queue stack bitset
//tuple cstdio numeric iterator algorithm cmath chrono cassert
using namespace std;    // :s/ /\r/g  :s/\w*/#include <\0>/g
#define REP(i,n)    for(auto i = decltype(n)(0); i<(n); i++)
#define all(x)      x.begin(), x.end()
using ll = long long; using ld = long double; using vi = vector<ll>;
const bool LOG = false; void Log() { if(LOG) cerr << "\n"; }
template<class T, class... S> void Log(T t, S... s){
    if(LOG) cerr << t << "\t", Log(s...); }
int main(){ ios::sync_with_stdio(false); cin.tie(nullptr); return 0; }
```

1.3 Java Template

```java
import java.io.OutputStream;
import java.io.InputStream;
import java.io.PrintWriter;
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.InputStream;
import java.io.IOException;

import java.util.Arrays;
import java.math.BigInteger;

public class Main { // Check what this should be called
    public static void main(String[] args) {
        InputReader in = new InputReader(System.in);
        PrintWriter out = new PrintWriter(System.out);
        Solver s = new Solver();
        s.solve(in, out);
        out.close();
    }

    static class Solver {
        public void solve(InputReader in, PrintWriter out) {
            // solve
        }
    }

    static class InputReader {
        public BufferedReader reader;
        public StringTokenizer tokenizer;
        public InputReader(InputStream st) {
            reader = new BufferedReader(new InputStreamReader(st), 32768);
```

```
            tokenizer = null;
        }
        public String next() {
            while (tokenizer == null || !tokenizer.hasMoreTokens()) {
                try {
                    String s = reader.readLine();
                    if (s == null) {
                        tokenizer = null; break; }
                    if (s.isEmpty()) continue;
                    tokenizer = new StringTokenizer(s);
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            return (tokenizer != null && tokenizer.hasMoreTokens()
                ? tokenizer.nextToken() : null);
        }
        public int nextInt() {
            String s = next();
            if (s != null) return Integer.parseInt(s);
            else return -1; // handle appropriately
        }
    }
}
```

2 Data Structures

2.1 Union Find

```
struct UnionFind {
    vi par, rank, size; int c;
    UnionFind(int n) : par(n), rank(n,0), size(n,1), c(n) {
        for (int i = 0; i < n; ++i) par[i] = i;
    }

    int find(int i) { return (par[i] == i ? i : (par[i] = find(par[i]))); }
    bool same(int i, int j) { return find(i) == find(j); }
    int get_size(int i) { return size[find(i)]; }
    int count() { return c; }

    void merge(int i, int j) {
        if ((i = find(i)) == (j = find(j))) return;
        c--;
        if (rank[i] > rank[j]) swap(i, j);
        par[i] = j; size[j] += size[i];
        if (rank[i] == rank[j]) rank[j]++;
    }
};
```

2.2 Max Queue

`dequeue` runs in amortized constant time. Can be modified to query minimum, gcd/lcm, set union/intersection (use bitmasks), etc.

```
template <class T>
class MaxQueue {
public:
    stack< pair<T, T> > inbox, outbox;
    void enqueue(T val) {
        T m = val;
        if (!inbox.empty()) m = max(m, inbox.top().second);
        inbox.push(pair<T, T>(val, m));
    }
    bool dequeue(T* d = nullptr) {
        if (outbox.empty() && !inbox.empty()) {
            pair<T, T> t = inbox.top(); inbox.pop();
            outbox.push(pair<T, T>(t.first, t.first));
            while (!inbox.empty()) {
                t = inbox.top(); inbox.pop();
                T m = max(t.first, outbox.top().second);
                outbox.push(pair<T, T>(t.first, m));
            }
        }
        if (outbox.empty()) return false;
        else {
            if (d != nullptr) *d = outbox.top().first;
            outbox.pop();
            return true;
        }
    }
    bool empty() { return outbox.empty() && inbox.empty(); }
    size_t size() { return outbox.size() + inbox.size(); }
    T get_max() {
        if (outbox.empty()) return inbox.top().second;
        if (inbox.empty()) return outbox.top().second;
        return max(outbox.top().second, inbox.top().second);
    }
};
```

2.3 Fenwick Tree

The tree is 1-based! Use indices $1..n$.

```
template <class T>
struct FenwickTree {        // use 1 based indices!!!
    int n; vector<T> tree;
    FenwickTree(int n) : n(n) { tree.assign(n + 1, 0); }
    T query(int l, int r) { return query(r) - query(l - 1); }
    T query(int r) {
        T s = 0;
        for(; r > 0; r -= (r & (-r))) s += tree[r];
        return s;
    }
    void update(int i, T v) {
        for(; i <= n; i += (i & (-i))) tree[i] += v;
    }
};
```

2.4 2D Fenwick Tree

Can easily be extended to any dimension.

```
template <class T>
struct FenwickTree2D {
```

```cpp
    vector< vector<T> > tree;
    int n;
    FenwickTree2D(int n) : n(n) { tree.assign(n + 1, vector<T>(n + 1, 0)); }
    T query(int x1, int y1, int x2, int y2) {
        return query(x2,y2)+query(x1-1,y1-1)-query(x2,y1-1)-query(x1-1,y2);
    }
    T query(int x, int y) {
        T s = 0;
        for (int i = x; i > 0; i -= (i & (-i)))
            for (int j = y; j > 0; j -= (j & (-j)))
                s += tree[i][j];
        return s;
    }
    void update(int x, int y, T v) {
        for (int i = x; i <= n; i += (i & (-i)))
            for (int j = y; j <= n; j += (j & (-j)))
                tree[i][j] += v;
    }
};
```

2.5 Sparse Table

For $O(1)$ range minimum query with $O(n \lg n)$ precalculation.

```cpp
using T = double; using vt = vector<T>; using vvt = vector<vt>;
struct SparseTable{
    vvt d;
    SparseTable(vt &a) : d(vvt{a}) {
        int N = a.size();
        for(auto s = 1; 2*s <= N; s *= 2){
            d.push_back(vt(N - 2*s + 1));
            auto &n = d.back(); auto &l = d[d.size()-2];
            for(int i = 0; i + 2*s<=N; ++i) n[i] = min(l[i], l[i+s]);
        }
    }
    int rmq(int l, int r){ // 0 <= l <= r < a.size()
        int p = 8*sizeof(int) - 1 - __builtin_clz(r+1-l);
        return min(d[p][l], d[p][r+1-(1<<p)]);
    }
};
```

2.6 Segment Tree

The range should be of the form 2^p.

```cpp
template <class T, T(*op)(T, T), T ident>
struct SegmentTree {
    int n; vector<T> tree;
    SegmentTree(vector<T> &init) : n(init.size()), tree(2 * n, ident) {
        copy(init.begin(), init.end(), tree.begin() + n);
        for (int j = n - 1; j > 0; --j)
            tree[j] = op(tree[2*j], tree[2*j+1]);
    }
    void update(int i, T val) {
        for (tree[i+n] = val, i = (i+n)/2; i > 0; i /= 2)
            tree[i] = op(tree[2*i], tree[2*i+1]);
    }
    T query(int l, int r) {
        T lhs = T(ident), rhs = T(ident);
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (  l&1 ) lhs = op(lhs, tree[l++]);
            if (!(r&1)) rhs = op(tree[r--], rhs);
        }
        return op(l == r ? op(lhs, tree[l]) : lhs, rhs);
    }
};
```

2.7 Lazy Dynamic Segment Tree

```cpp
using T=ll; using U=ll;                             // exclusive right bounds
T t_id; U u_id;
T op(T a, T b){ return a+b; }
void join(U &a, U b){ a+=b; }
void apply(T &t, U u, int x){ t+=x*u; }
T part(T t, int r, int p){ return t/r*p; }
struct DynamicSegmentTree {
    struct Node { int l, r, lc, rc; T t; U u;
        Node(int l, int r):l(l),r(r),lc(-1),rc(-1),t(t_id),u(u_id){}
    };
    vector<Node> tree;
    DynamicSegmentTree(int N) { tree.push_back({0,N}); }
    void push(Node &n, U u){ apply(n.t, u, n.r-n.l); join(n.u,u); }
    void push(Node &n){push(tree[n.lc],n.u);push(tree[n.rc],n.u);n.u=u_id;}
    T query(int l, int r, int i = 0) { auto &n = tree[i];
        if(r <= n.l || n.r <= l) return t_id;
        if(l <= n.l && n.r <= r) return n.t;
        if(n.lc < 0) return part(n.t, n.r-n.l, min(n.r,r)-max(n.l,l));
        return push(n), op(query(l,r,n.lc),query(l,r,n.rc));
    }
    void update(int l, int r, U u, int i = 0) { auto &n = tree[i];
        if(r <= n.l || n.r <= l) return;
        if(l <= n.l && n.r <= r) return push(n,u);
        if(n.lc < 0) { int m = (n.l + n.r) / 2;
            n.lc = tree.size();             n.rc = n.lc+1;
            tree.push_back({tree[i].l, m}); tree.push_back({m, tree[i].r});
        }
        push(tree[i]); update(l,r,u,tree[i].lc); update(l,r,u,tree[i].rc);
        tree[i].t = op(tree[tree[i].lc].t, tree[tree[i].rc].t);
    }
};
```

2.8 Implicit Cartesian Tree

The indices are zero-based. Also, don't forget to initialise the empty tree to `NULL`. (Pretty much) all operations take $O(\log n)$ time.

```cpp
struct Node {
    ll val, mx;
    int size, priority;
    bool rev = false;
    Node *l = NULL, *r = NULL;
    Node(ll _val) : val(_val), mx(_val), size(1) { priority = rand(); }
};
```

```
int size(Node *p) { return p == NULL ? 0 : p->size; }
ll getmax(Node *p) { return p == NULL ? -LLINF : p->mx; }
void update(Node *p) {
    if (p == NULL) return;
    p->size = 1 + size(p->l) + size(p->r);
    p->mx = max(p->val, max(getmax(p->l), getmax(p->r)));
}
void propagate(Node *p) {
    if (p == NULL || !p->rev) return;
    swap(p->l, p->r);
    if (p->l != NULL) p->l->rev ^= true;
    if (p->r != NULL) p->r->rev ^= true;
    p->rev = false;
}
void merge(Node *&t, Node *l, Node *r) {
    propagate(l); propagate(r);
    if (l == NULL)      { t = r; }
    else if (r == NULL) { t = l; }
    else if (l->priority > r->priority) {
        merge(l->r, l->r, r); t = l; }
    else { merge(r->l, l, r->l); t = r; }
    update(t);
}
void split(Node *t, Node *&l, Node *&r, int at) {
    propagate(t);
    if (t == NULL) { l = r = NULL; return; }
    int id = size(t->l) + 1;
    if (id > at) { split(t->l, l, t->l, at); r = t; }
    else { split(t->r, t->r, r, at - id); l = t; }
    update(t);
}
void insert(Node *&t, ll val, int pos) {
    propagate(t);
    Node *n = new Node(val), *l, *r;
    split(t, l, r, pos);
    merge(t, l, n);
    merge(t, t, r);
}
void erase(Node *&t, int pos, bool del = true) {
    propagate(t);
    Node *L, *rm;
    split(t, t, L, pos);
    split(L, rm, L, 1);
    merge(t, t, L);
    if (del && rm != NULL) delete rm;
}
void reverse(Node *t, int l, int r) {
    propagate(t);
    Node *L, *R;
    split(t, t, L, l);
    split(L, L, R, r - l + 1);
    if (L != NULL) L->rev = true;
    merge(t, t, L);
    merge(t, t, R);
}
ll at(Node *t, int pos) {
    propagate(t);
    int id = size(t->l);
    if (pos == id) return t->val;
    else if (ps > id) return at(t->r, pos - id - 1);
    else return at(t->l, pos);
}
ll range_maximum(Node *t, int l, int r) {
    propagate(t);
    Node *L, *R;
    split(t, t, L, l);
    split(L, L, R, r - l + 1);
    ll ret = getmax(L);
    merge(t, t, L);
    merge(t, t, R);
    return ret;
}
void cleanup(Node *p) {
    if (p == NULL) return;
    cleanup(p->l); cleanup(p->r);
    delete p;
}
```

2.9 KD tree

```
struct P{ ll x,y; };
struct Box{
    ll xl, xh, yl, yh;
    Box(ll xl=-LLINF, ll xh=-LLINF, ll yl=LLINF, ll yh=LLINF) :
        xl(xl), xh(xh), yl(yl), yh(yh) {}
    bool contains(const P &p) const {
        return xl <= p.x && p.x <= xh && yl <= p.y && p.y <= yh;
    }
    bool contains(const Box &b) const {
        return xl <= b.xl && b.xh <= xh && yl <= b.yl && b.yh <= yh;
    }
    bool disjunct(const Box &b) const {
        return xh < b.xl || b.xh < xl || yh < b.yl || b.yh < yl;
    }
};
struct Node {
    ll i, cl, cr; bool hz;
    Node(ll i, bool h) : i{i}, cl{-1}, cr{-1}, hz{h} {};
};
struct KDTree {
    vector<P> &ps; vector<Node> tree;
    KDTree(vector<P> &ps) : ps{ps} {
        vi x(ps.size()); iota(x.begin(), x.end(), 0); vi y(x);
        sort(x.begin(), x.end(), [&](ll l, ll r){ return compx(l,r); });
        sort(y.begin(), y.end(), [&](ll l, ll r){ return compy(l,r); });
        tree.reserve(ps.size());
        build(x, y, true);
    }
bool compx(ll l,ll r){return tie(ps[l].x,ps[l].y,l)<tie(ps[r].x,ps[r].y,r);}
bool compy(ll l,ll r){return tie(ps[l].y,ps[l].x,l)<tie(ps[r].y,ps[r].x,r);}
    int build(vi &x, vi &y, bool h){
        if(x.size()==0) return -1;
        ll m = x.size()/2, n = tree.size();
        vi xl, xh, yl, yh;
        if(h){ // horizontal
```

```
            ll s = x[m]; tree.push_back({s, h});
            xh.assign(x.begin()+m+1, x.end()), xl = move(x);
            xl.resize(m);
            for(const auto &p : y)
                if(p==s) continue;
                else if(compx(p,s)) yl.push_back(p);
                else                yh.push_back(p);
        } else { // vertical
            ll s = y[m]; tree.push_back({s, h});
            yh.assign(y.begin()+m+1, y.end()), yl = move(y);
            yl.resize(m);
            for(const auto &p : x)
                if(p==s) continue;
                else if(compy(p,s)) xl.push_back(p);
                else                xh.push_back(p);
        }
        tree[n].cl = build(xl,yl,!h); tree[n].cr = build(xh,yh,!h);
        return n;
    }
    vi ans;     // returns a list of indices in ps
    vi query(const Box &q){ ans.clear(); query(q, Box(), 0); return ans; }
    void query(const Box &q, const Box &b, ll n){
        auto &node = tree[n]; auto &p = ps[node.i];
        if(q.contains(b)){ allq(n); return; }
        if(q.disjunct(b)) return;
        if(q.contains(p)) ans.push_back(node.i);
        Box b1=b, b2=b;
        if(node.hz) b1.xh = b2.xl = p.x;
        else        b1.yh = b2.yl = p.y;
        query(q,b1,node.cl); query(q,b2,node.cr);
    }
    void allq(ll n){ if(n==-1) return;
        ans.push_back(tree[n].i); allq(tree[n].cl); allq(tree[n].cr);
    }
};
```

2.10 AVL Tree

Can be augmented to support in $O(\log n)$ time: range queries/updates (similar to a segment tree), insert at position n/query for position n, order statistics, etc.

```
template <class T>
struct AVL_Tree {
    struct AVL_Node {
        T val;
        AVL_Node *p, *l, *r;
        int size, height;
        AVL_Node(T &_val, AVL_Node *_p = NULL)
         : val(_val), p(_p), l(NULL), r(NULL), size(1), height(0) { }
    };
    AVL_Node *root;
    AVL_Tree() : root(NULL) { }

    // Querying
    AVL_Node *find(T &key) { // O(lg n)
        AVL_Node *c = root;
        while (c != NULL && c->val != key) {
            if (c->val < key) c = c->r;
            else c = c->l;
        }
        return c;
    }
    // maximum and predecessor can be written in a similar manner
    AVL_Node *minimum(AVL_Node *n) { // O(lg n)
        if (n != NULL) while (n->l != NULL) n = n->l; return n;
    }
    AVL_Node *minimum() { return minimum(root); } // O(lg n)
    AVL_Node *successor(AVL_Node *n) { // O(lg n)
        if (n->r != NULL) return minimum(n->r);
        AVL_Node *p = n->p;
        while (p != NULL && n == p->r) { n = p; p = n->p; }
        return p;
    }

    // Modification
    AVL_Node *insert(T &nval) { // O(lg n)
        AVL_Node *p = NULL, *c = root;
        while (c != NULL) {
            p = c;
            c = (c->val < nval ? c->r : c->l);
        }
        AVL_Node *r = new AVL_Node(nval, p);
        (p == NULL ? root : (
            nval < p->val ? p->l : p->r)) = r;
        _fixup(r);
        return r;
    }
    void remove(AVL_Node *n, bool del = true) { // O(lg n)
        if (n == NULL) return;
        if (n->l != NULL && n->r != NULL) {
            AVL_Node *y = successor(n), *z = y->par;
            if (z != n) {
                _transplant(y, y->r);
                y->r = n->r;
                y->r->p = y;
            }
            _transplant(n, y);
            y->l = n->l;
            y->l->p = y;
            _fixup(z->r == NULL ? z : z->r);
            if (del) delete n;
            return;
        } else if (n->l != NULL) {
            _pchild(n) = n->l;
            n->l->p = n->p;
        } else if (n->r != NULL) {
            _pchild(n) = n->r;
            n->r->p = n->p;
        } else _pchild(n) = NULL;
        _fixup(n->p);
        if (del) delete n;
    }
    void cleanup() { _cleanup(root); }

    // Helpers
    void _transplant(AVL_Node *u, AVL_Node *v) {
```

```
        _pchild(u) = v;
        if (v != NULL) v->p = u->p;
    }
    AVL_Node *&_pchild(AVL_Node *n) {
        return (n == NULL ? root : (n->p == NULL ? root :
            (n->p->l == n ? n->p->l : n->p->r)));
    }
    void _augmentation(AVL_Node *n) {
        if (n == NULL) return;
        n->height = 1 + max(_get_height(n->l), _get_height(n->r));
        n->size = 1 + _get_size(n->l) + _get_size(n->r);
    }
    int _get_height(AVL_Node *n) { return (n == NULL ? 0 : n->height); }
    int _get_size(AVL_Node *n) { return (n == NULL ? 0 : n->size); }
    bool _balanced(AVL_Node *n) {
        return (abs(_get_height(n->l) - _get_height(n->r)) <= 1);
    }
    bool _leans_left(AVL_Node *n) {
        return _get_height(n->l) > _get_height(n->r);
    }
    bool _leans_right(AVL_Node *n) {
        return _get_height(n->r) > _get_height(n->l);
    }
#define ROTATE(L, R) \
    AVL_Node *o = n->R; \
    n->R = o->L; \
    if (o->L != NULL) o->L->p = n; \
    o->p = n->p; \
    _pchild(n) = o; \
    o->L = n; \
    n->p = o; \
    _augmentation(n); \
    _augmentation(o);
    void _left_rotate(AVL_Node *n) { ROTATE(l, r); }
    void _right_rotate(AVL_Node *n) { ROTATE(r, l); }
    void _fixup(AVL_Node *n) {
        while (n != NULL) {
            _augmentation(n);
            if (!_balanced(n)) {
                if (_leans_left(n)&&_leans_right(n->l)) _left_rotate(n->l);
                else if (_leans_right(n) && _leans_left(n->r))
                    _right_rotate(n->r);
                if (_leans_left(n)) _right_rotate(n);
                if (_leans_right(n)) _left_rotate(n);
            }
            n = n->p;
        }
    }
    void _cleanup(AVL_Node *n) {
        if (n->l != NULL) _cleanup(n->l);
        if (n->r != NULL) _cleanup(n->r);
    }
};
```

2.11 Treap

Can be used like the built-in **set**, except that it also supports order statistics, can be merged/split in $O(\log n)$ time, can support range queries, and more.

```
struct Node {
    ll val;
    int size, priority;
    Node *l = NULL, *r = NULL;
    Node(ll _v) : val(_v), size(1) { priority = rand(); }
};

int size(Node *p) { return p == NULL ? 0 : p->size; }
void update(Node *p) {
    if (p == NULL) return;
    p->size = 1 + size(p->l) + size(p->r);
}
void merge(Node *&t, Node *l, Node *r) {
    if (l == NULL)      { t = r; }
    else if (r == NULL) { t = l; }
    else if (l->priority > r->priority) {
        merge(l->r, l->r, r); t = l;
    } else {
        merge(r->l, l, r->l); t = r;
    } update(t);
}
void split(Node *t, Node *&l, Node *&r, ll val) {
    if (t == NULL) { l = r = NULL; return; }
    if (t->val >= val) { // val goes with the right set
        split(t->l, l, t->l, val); r = t;
    } else {
        split(t->r, t->r, r, val); l = t;
    } update(t);
}
bool insert(Node *&t, ll val) {
    // returns false if the element already existed
    Node *n = new Node(val), *l, *r;
    split(t, l, t, val);
    split(t, t, r, val + 1);
    bool empty = (t == NULL);
    merge(t, l, n);
    merge(t, t, r);
    return empty;
}
void erase(Node *&t, ll val, bool del = true) {
    // returns false if the element did not exist
    Node *l, *rm;
    split(t, l, t, val);
    split(t, rm, t, val + 1);
    bool exists = (t != NULL);
    merge(t, l, t);
    if (del && rm != NULL) delete rm;
    return exists;
}
void cleanup(Node *p) {
    if (p == NULL) return;
    cleanup(p->l); cleanup(p->r);
    delete p;
```

```
}
```

2.12 Prefix Trie

```
const int ALPHABET_SIZE = 26;
inline int mp(char c) { return c - 'a'; }

struct Node {
    Node* ch[ALPHABET_SIZE];
    bool isleaf = false;
    Node() {
        for(int i = 0; i < ALPHABET_SIZE; ++i) ch[i] = nullptr;
    }

    void insert(string &s, int i = 0) {
        if (i == s.length()) isleaf = true;
        else {
            int v = mp(s[i]);
            if (ch[v] == nullptr)
                ch[v] = new Node();
            ch[v]->insert(s, i + 1);
        }
    }

    bool contains(string &s, int i = 0) {
        if (i == s.length()) return isleaf;
        else {
            int v = mp(s[i]);
            if (ch[v] == nullptr) return false;
            else return ch[v]->contains(s, i + 1);
        }
    }

    void cleanup() {
        for (int i = 0; i < ALPHABET_SIZE; ++i)
            if (ch[i] != nullptr) {
                ch[i]->cleanup();
                delete ch[i];
            }
    }
};
```

2.13 Suffix Array

Note: dont forget to invert the returned array. **Complexity:** $O(n \log n)$

```
    string s;
    int n;
    vvi P;
    SuffixArray(string &_s) : s(_s), n(_s.length()) { construct(); }
    void construct() {
        P.push_back(vi(n, 0));
        compress();
        vi occ(n + 1, 0), s1(n, 0), s2(n, 0);
        for (int k = 1, cnt = 1; cnt / 2 < n; ++k, cnt *= 2) {
            P.push_back(vi(n, 0));
            fill(occ.begin(), occ.end(), 0);
            for (int i = 0; i < n; ++i)
                occ[i+cnt<n ? P[k-1][i+cnt]+1 : 0]++;
            partial_sum(occ.begin(), occ.end(), occ.begin());
            for (int i = n - 1; i >= 0; --i)
                s1[--occ[i+cnt<n ? P[k-1][i+cnt]+1 : 0]] = i;
            fill(occ.begin(), occ.end(), 0);
            for (int i = 0; i < n; ++i)
                occ[P[k-1][s1[i]]]++;
            partial_sum(occ.begin(), occ.end(), occ.begin());
            for (int i = n - 1; i >= 0; --i)
                s2[--occ[P[k-1][s1[i]]]] = s1[i];
            for (int i = 1; i < n; ++i) {
                P[k][s2[i]] = same(s2[i], s2[i - 1], k, cnt)
                    ? P[k][s2[i - 1]] : i;
            }
        }
    }
    bool same(int i, int j, int k, int l) {
        return P[k - 1][i] == P[k - 1][j]
            && (i + l < n ? P[k - 1][i + l] : -1)
            == (j + l < n ? P[k - 1][j + l] : -1);
    }
    void compress() {
        vi cnt(256, 0);
        for (int i = 0; i < n; ++i) cnt[s[i]]++;
        for (int i = 0, mp = 0; i < 256; ++i)
            if (cnt[i] > 0) cnt[i] = mp++;
        for (int i = 0; i < n; ++i) P[0][i] = cnt[s[i]];
    }
    vi &get_array() { return P.back(); }
    int lcp(int x, int y) {
        int ret = 0;
        if (x == y) return n - x;
        for (int k = P.size() - 1; k >= 0 && x < n && y < n; --k)
            if (P[k][x] == P[k][y]) {
                x += 1 << k;
                y += 1 << k;
                ret += 1 << k;
            }
        return ret;
    }
};
```

2.14 Suffix Tree

Complexity: $O(n)$

```
using T = char;
using M = map<T,int>;       // or array<T,ALPHABET_SIZE>
using V = string;           // could be vector<T> as well
using It = V::const_iterator;
struct Node{
    It b, e; M edges; int link;     // end is exclusive
    Node(It b, It e) : b(b), e(e), link(-1) {}
    int size() const { return e-b; }
};
struct SuffixTree{
    const V &s; vector<Node> t;
```

```
    int root,n,len,remainder,llink; It edge;
    SuffixTree(const V &s) : s(s) { build(); }
    int add_node(It b, It e){ return t.push_back({b,e}), t.size()-1; }
    int add_node(It b){ return add_node(b,s.end()); }
    void link(int node){ if(llink) t[llink].link = node; llink = node; }
    void build(){
        len = remainder =  0; edge = s.begin();
        n = root = add_node(s.begin(), s.begin());
        for(auto i = s.begin(); i != s.end(); ++i){
            ++remainder; llink = 0;
            while(remainder){
                if(len == 0) edge = i;
                if(t[n].edges[*edge] == 0){          // add new leaf
                    t[n].edges[*edge] = add_node(i); link(n);
                } else {
                    auto x = t[n].edges[*edge];      // neXt node [with edge]
                    if(len >= t[x].size()){          // walk to next node
                        len -= t[x].size(); edge += t[x].size(); n = x;
                        continue;
                    }
                    if(*(t[x].b + len) == *i){       // walk along edge
                        ++len; link(n); break;
                    }                                // split edge
                    auto split = add_node(t[x].b, t[x].b+len);
                    t[n].edges[*edge] = split;
                    t[x].b += len;
                    t[split].edges[*i] = add_node(i);
                    t[split].edges[*t[x].b] = x;
                    link(split);
                }
                --remainder;
                if(n == root && len > 0)
                    --len, edge = i - remainder + 1;
                else n = t[n].link > 0 ? t[n].link : root;
            }
        }
    }
};
```

2.15 Suffix Automaton

Complexity: $O(n)$

```
using T = char; using M = map<T,int>; using V = string;
struct Node {         // s: start, len: length, link: suffix link, e: edges
    int s, len, link; M e; bool term;                // term: terminal node?
    Node(int s, int len, int link=-1):s(s), len(len), link(link), term(0) {}
};
struct SuffixAutomaton{
    const V &s; vector<Node> t; int l;  // string; tree; last added state
    SuffixAutomaton(const V &s) : s(s) { build(); }
    void build(){
        l = t.size(); t.push_back({0,-1});               // root node
        for(auto c : s){
            int p=l, x=t.size(); t.push_back({0,t[l].len + 1}); // new node
            while(p>=0 && t[p].e[c] == 0) t[p].e[c] = x,  p= t[p].link;
            if(p<0) t[x].link = 0;                       // at root
            else {
                int q = t[p].e[c];                       // the c-child of q
                if(t[q].len == t[p].len + 1) t[x].link = q;
                else {                                   // cloning of q
                    int cl = t.size(); t.push_back(t[q]);
                    t[cl].len = t[p].len + 1;
                    t[cl].s = t[q].s + t[q].len - t[p].len - 1;
                    t[x].link = t[q].link = cl;
                    while(p >= 0 && t[p].e.count(c) > 0 && t[p].e[c] == q)
                        t[p].e[c] = cl, p = t[p].link;  // relink suffix
                }
            }
            l = x;                                       // update last
        }
        while(l>=0) t[l].term = true, l = t[l].link;
    }
};
```

2.16 Built-in datastructures

```
// Minimum Heap
#include <queue>
template<class T>
using min_queue = priority_queue<T, vector<T>, greater<T>>;

// Order Statistics Tree
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class TIn, class TOut>
using order_tree = tree<
    TIn, TOut, less<TIn>, // key, value types. TOut can be null_type
    rb_tree_tag, tree_order_statistics_node_update>;
// find_by_order(int r) (0-based)
// order_of_key(TIn v)
// use key pair<Tin,int> {value, counter} for multiset/multimap
```

3 Basic Graph algorithms

3.1 Edge Classification

Complexity: $O(V + E)$

```
struct Edge_Classification {
    vector<vi> &edges; int V; vi color, parent;
    Edge_Classification(vector<vi> &edges) :
        edges(edges), V(edges.size()),
        color(V,-1), parent(V, -1) {}

    void visit(int u) {
        color[u] = 1;       // in progress
        for (int v : edges[u]) {
            if (color[v] == -1) { // u -> v is a tree edge
                parent[v] = u;
                visit(v);
            } else if (color[v] == 1) {
                if (v == parent[u]) {} // u -> v is a bidirectional edge
                else {} // u -> v is a back edge (thus contained in a cycle)
```

```
            } else if (color[v] == 2) {} // u -> v is a forward/cross edge
        }
        color[u] = 2;        // done
    }
    void run(){
        for (int u = 0; u < V; ++u) if(color[u] < 0) visit(u);
    }
};
```

3.2 Topological sort

Complexity: $O(V + E)$

```
struct Toposort {
    vector<vi> &edges;
    int V, s_ix; // sorted-index
    vi sorted, visited;

    Toposort(vector<vi> &edges) :
        edges(edges), V(edges.size()), s_ix(V),
        sorted(V,-1), visited(V,false) {}

    void visit(int u) {
        visited[u] = true;
        for (int v : edges[u])
            if (!visited[v]) visit(v);
        sorted[--s_ix] = u;
    }
    void topo_sort() {
        for (int i = 0; i < V; ++i) if (!visited[i]) visit(i);
    }
};
```

3.3 Tarjan: SCCs

Complexity: $O(V + E)$

```
struct Tarjan {
    vvi &edges;
    int V, counter = 0, C = 0;
    vi n, l;
    vb vs;
    stack<int> st;

    Tarjan(vvi &e) : edges(e), V(e.size()),
        n(V, -1), l(V, -1), vs(V, false) { }

    void visit(int u, vi &com) {
        l[u] = n[u] = counter++;
        st.push(u); vs[u] = true;
        for (auto &&v : edges[u]) {
            if (n[v] == -1) visit(v, com);
            if (vs[v]) l[u] = min(l[u], l[v]);
        }
        if (l[u] == n[u]) {
            while (true) {
                int v = st.top(); st.pop(); vs[v] = false;
                com[v] = C;     //<== ACT HERE
                if (u == v) break;
            }
            C++;
        }
    }

    int find_sccs(vi &com) { // component indices will be stored in 'com'
        com.assign(V, -1);
        C = 0;
        for (int u = 0; u < V; ++u)
            if (n[u] == -1) visit(u, com);
        return C;
    }

    // scc is a map of the original vertices of the graph
    // to the vertices of the SCC graph, scc_graph is its
    // adjacency list.
    // Scc indices and edges are stored in 'scc' and 'scc_graph'.
    void scc_collapse(vi &scc, vvi &scc_graph) {
        find_sccs(scc);
        scc_graph.assign(C,vi());
        set<ii> rec; // recorded edges
        for (int u = 0; u < V; ++u) {
            assert(scc[u] != -1);
            for (int v : edges[u]) {
                if (scc[v] == scc[u] ||
                    rec.find({scc[u], scc[v]}) != rec.end()) continue;
                scc_graph[scc[u]].push_back(scc[v]);
                rec.insert({scc[u], scc[v]});
            }
        }
    }
};
```

3.4 Biconnected components

Complexity: $O(V + E)$

```
struct BCC{     // find AVs and bridges in an undirected graph
    vvi &edges;
    int V, counter = 0, root, rcs;      // root and # children of root
    vi n,l;                             // nodes,low
    stack<int> s;
    BCC(vvi &e) : edges(e), V(e.size()), n(V,-1), l(V,-1) {}
    void visit(int u, int p) {          // also pass the parent
        l[u] = n[u] = counter++; s.push(u);
        for(auto &v : edges[u]){
            if (n[v] == -1) {
                if (u == root) rcs++; visit(v,u);
                if (l[v] >= n[u]) {}    // u is an articulation point
                if (l[v] > n[u]) {      // u<->v is a bridge
                    while(true){        // biconnected component
                        int w = s.top(); s.pop();   // <= ACT HERE
                        if(w==v) break;
                    }
                }
                l[u] = min(l[u], l[v]);
```

```
            } else if (v != p) l[u] = min(l[u], n[v]);
        }
    }
    void run() {
        for (int u = 0; u < V; ++u) if (n[u] == -1) {
            root = u; rcs = 0; visit(u,-1);
            if(rcs > 1) {}              // u is articulation point
        }
    }
};
```

3.5 Kruskal's algorithm

Complexity: $O(E \log V)$ **Dependencies:** Union Find

```
#include "../datastructures/unionfind.cpp"
// Edges are given as (weight, (u, v)) triples.
struct E {int u, v, weight;};
bool operator<(const E &l, const E &r){return l.weight < r.weight;}
int kruskal(vector<E> &edges, int V) {
    sort(edges.begin(), edges.end());
    int cost = 0, count = 0;
    UnionFind uf(V);
    for (auto &e : edges) {
        if (!uf.same(e.u, e.v)) {
            // (w, (u, v)) is part of the MST
            cost += e.weight;
            uf.union_set(e.u, e.v);
            if ((++count) == V - 1) break;
        }
    }
    return cost;
}
```

3.6 Prim's algorithm

Complexity: $O(E \log V)$

```
struct AdjEdge { int v; ll weight; };   // adjacency list edge
struct Edge { int u, v; };            // edge u->v for output
struct PQ { ll weight; Edge e; };    // PQ element
bool operator>(const PQ &l, const PQ &r) { return l.weight > r.weight; }
ll prim(vector<vector<AdjEdge>> &adj, vector<Edge> &tree) {
    ll tc = 0; vb intree(adj.size(), false);
    priority_queue<PQ, vector<PQ>, greater<PQ> > pq;
    intree[0] = true;
    for (auto &e : adj[0]) pq.push({e.weight, {0, e.v}});
    while (!pq.empty()) {
        auto &top = pq.top();
        ll c = top.weight; auto e = top.e; pq.pop();
        if (intree[e.v]) continue;
        intree[e.v] = true; tc += c; tree.push_back(e);
        for (auto &e2 : adj[e.v])
            if (!intree[e2.v]) pq.push({e2.weight, {e.v, e2.v}});
    }
    return tc;
}
```

3.7 Dijkstra's algorithm

Complexity: $O((V + E) \log V)$

```
struct Edge{ int v; ll weight; };   // input edges
struct PQ{ ll d; int v; };          // distance and target
bool operator>(const PQ &l, const PQ &r){ return l.d > r.d; }
ll dijkstra(vector<vector<Edge>> &edges, int s, int t) {
    vector<ll> dist(edges.size(),LLINF);
    priority_queue<PQ,vector<PQ>,greater<PQ>> pq;
    dist[s] = 0; pq.push({0, s});
    while (!pq.empty()) {
        auto d = pq.top().d; auto u = pq.top().v; pq.pop();
        if(u==t) break;         // target reached
        if (d == dist[u])
            for(auto &e : edges[u]) if (dist[e.v] > d + e.weight)
                pq.push({dist[e.v] = d + e.weight, e.v});
    }
    return dist[t];
}
```

3.8 Bellman-Ford

An improved (but slower) version of Bellmann-Ford that can indicate for each vertex separately whether it is reachable, and if so, whether there is a lowerbound on the length of the shortest path. **Complexity:** $O(VE)$

```
void bellmann_ford_extended(vvii &e, int source, vi &dist, vb &cyc) {
    dist.assign(e.size(), INF);
    cyc.assign(e.size(), false); // true when u is in a <0 cycle
    dist[source] = 0;
    for (int iter = 0; iter < e.size() - 1; ++iter){
        bool relax = false;
        for (int u = 0; u < e.size(); ++u)
            if (dist[u] == INF) continue;
            else for (auto &e : e[u])
                if(dist[u]+e.second < dist[e.first])
                    dist[e.first] = dist[u]+e.second, relax = true;
        if(!relax) break;
    }
    bool ch = true;
    while (ch) {                // keep going untill no more changes
        ch = false;             // set dist to -INF when in cycle
        for (int u = 0; u < e.size(); ++u)
            if (dist[u] == INF) continue;
            else for (auto &e : e[u])
                if (dist[e.first] > dist[u] + e.second
                    && !cyc[e.first]) {
                    dist[e.first] = -INF;
                    ch = true;      //return true for cycle detection only
                    cyc[e.first] = true;
                }
    }
}
```

3.9 Floyd-Warshall algorithm

Transitive closure: `R[a,c] = R[a,c] | (R[a,b] & R[b,c]))`, transitive reduction: `R[a,c] = R[a,c] & !(R[a,b] & R[b, c])`. **Complexity:** $O(V^3)$

```
// adj should be a V*V array s.t. adj[i][j] contains the weight of
// the edge from i to j, INF if it does not exist.
// set adj[i][i] to 0; and always do adj[i][j] = min(adj[i][j], w)
int adj[100][100];
void floyd_warshall(int V) {
    for (int b = 0; b < V; ++b)
        for (int a = 0; a < V; ++a)
            for (int c = 0; c < V; ++c)
                if(adj[a][b] != INF && adj[b][c] != INF)
                    adj[a][c] = min(adj[a][c], adj[a][b] + adj[b][c]);
}
void setnegcycle(int V){        // set all -Infinity distances
    REP(a,V) REP(b,V) REP(c,V)              //tested on Kattis
        if(adj[a][c] != INF && adj[c][b] != INF && adj[c][c]<0){
            adj[a][b] = - INF;
            break;
        }
}
```

3.10 Johnson's reweighting

Apply Bellman-Ford to the graph with `d[u] = 0` (as if an extra vertex with zero weight edges were added), then reweight edges to $w_{uv} + h_u - h_v$, then use Dijkstra. **Complexity:** $O(VE \log V)$

3.11 Hierholzer's algorithm

Verify existence of the circuit/trail in advance (see Theorems in Graph Theory for more information). When looking for a trail, be sure to specify the starting vertex. **Complexity:** $O(V + E)$

```
struct edge {
    int v;
    list<edge>::iterator rev;
    edge(int _v) : v(_v) {};
};

void add_edge(vector< list<edge> > &adj, int u, int v) {
    adj[u].push_front(edge(v));
    adj[v].push_front(edge(u));
    adj[u].begin()->rev = adj[v].begin();
    adj[v].begin()->rev = adj[u].begin();
}

void remove_edge(vector< list<edge> > &adj, int s, list<edge>::iterator e) {
    adj[e->v].erase(e->rev);
    adj[s].erase(e);
}

eulerian_circuit(vector< list<edge> > &adj, vi &c, int start = 0) {
    stack<int> st;
    st.push(start);

    while(!st.empty()) {
        int u = st.top().first;
        if (adj[u].empty()) {
            c.push_back(u);
            st.pop();
        } else {
            st.push(adj[u].front().v);
            remove_edge(adj, u, adj[u].begin());
        }
    }
}
```

3.12 Bron-Kerbosch

Count the number of maximal cliques in a graph with up to a few hundred nodes. **Complexity:** $O(3^{n/3})$

```
constexpr size_t M = 128; using S = bitset<M>;
// count maximal cliques. Call with R=0, X=0, P[u]=1 forall u
int BronKerbosch(const vector<S> &edges, S &R, S &&P, S &&X){
    if(P.count() == 0 && X.count() == 0) return 1;
    auto PX = P | X; int p=-1;  // the last true bit is the pivot
    for(int i = M-1; i>=0; i--) if(PX[i]){ p = i; break; }
    auto mask = P & (~edges[p]); int count = 0;
    for (size_t u = 0; u < edges.size(); ++u) {
        if(!mask[u]) continue;
        R[u]=true;
        count += BronKerbosch(edges,R,P & edges[u],X & edges[u]);
        if(count > 1000) return count;
        R[u]=false; X[u]=true; P[u]=false;
    }
    return count;
}
```

3.13 Theorems in Graph Theory

Dilworth's theorem : The minimum number of disjoint chains into which S can be decomposed equals the length of a longest antichain of S.

Compute by defining a bipartite graph with a source u_x and sink v_x for each vertex x, and adding an edge (u_x, v_y) if $x \leq y, x \neq y$. Let m denote the size of the maximum matching, then the number of disjoint chains is $|S| - m$ (the collection of unmatched endpoints).

Mirsky's theorem : The minimum number of disjoint antichains into which S can be decomposed equals the length of a longest chain of S.

Compute by defining L_v to be the length of the longest chain ending at v. Sort S topologically and use bottom-up DP to compute L_u for all $u \in S$.

Kirchhoff's theorem : Define a $V \times V$ matrix M as: $M_{ij} = deg(i)$ if $i == j$, $M_{ij} = -1$ if $\{i, j\} \in E$, $M_{ij} = 0$ otherwise. Then the number of distinct spanning trees equals any minor of M.

Acyclicity : A directed graph is acyclic if and only if a depth-first search yields no back edges.

Euler Circuits and Trails : In an *undirected graph*, an *Eulerian Circuit* exists if and only if all vertices have even degree, and all vertices of nonzero degree belong to a single connected component. In an *undirected graph*, an *Eulerian Trail* exists if and only if at most two vertices have odd degree, and all of its vertices of nonzero degree belong to a single connected component. In a *directed graph*, an *Eulerian Circuit* exists if and only if every vertex has equal indegree and outdegree, and all vertices of nonzero degree belong to a single strongly connected component. In a *directed graph*, an *Eulerian Trail* exists if and only at most one vertex has $outdegree - indegree = 1$, at most one vertex has $indegree - outdegree = 1$, every other vertex has equal indegree and outdegree, and all vertices of nonzero degree belong to a single strongly connected component *in the underlying undirected graph*.

3.14 Centroid Decomposition

In case it is necessary to work with the subtrees directly, consider timestamping each node during the decomposition **Complexity:** $O(n \log n)$

```
struct CentroidDecomposition {
    vvi &e;         // The original tree
    vb tocheck;     // Used during decomposition
    vi size, p;
    int root;       // The decomposition
    vvi cd;
    CentroidDecomposition(vvi &tree) : e(tree) {
        int V = e.size();           // create initializer list?
        tocheck.assign(V, true);
        cd.assign(V, vi());
        p.assign(V, -1);
        size.assign(V, 0);

        dfs(0);
        root = decompose(0, V);
    }

    void dfs(int u) {
        for (int v : e[u]) {
            if (v == p[u]) continue;
            p[v] = u;
            dfs(v);
            size[u] += 1 + size[v];
        }
    }

    int decompose(int _u, int V) {
        // Find centroid
        int u = _u;
        while (true) {
            int nu = -1;
            for (int v : e[u]) {
                if (!tocheck[v] || v == p[u])
                    continue;
                if (1 + size[v] > V / 2) nu = v;
            }
            if (V - 1 - size[u] > V / 2 && p[u] != -1
                && tocheck[p[u]]) nu = p[u];
            if (nu != -1) u = nu; else break;
        }
        // Fix the sizes of the parents of the centroid
        for (int v = p[u]; v != -1 && tocheck[v]; v = p[v])
            size[v] -= 1 + size[u];
        // Find centroid children
        tocheck[u] = false;
        for (int v : e[u]) {
            if (!tocheck[v]) continue;
            int V2 = 1 + size[v];
            if (v == p[u]) V2 = V - 1 - size[u];
            cd[u].push_back(decompose(v, V2));
        }
        return u;
    }
};
```

3.15 Heavy-Light decomposition

Complexity: $O(n)$

```
struct HLD {
    int V; vvi &graph; // graph can be graph or childs only
    vi p, r, d, h; // parents, path-root; heavy child, depth
    HLD(vvi &graph, int root = 0) : V(graph.size()), graph(graph),
    p(V,-1), r(V,-1), d(V,0), h(V,-1) { dfs(root);
        for(int i=0; i<V; ++i) if (p[i]==-1 || h[p[i]]!=i)
            for (int j=i; j!=-1; j=h[j]) r[j] = i;
    }
    int dfs(int u){
        ii best={-1,-1}; int s=1, ss;   // best, size (of subtree)
        for(auto &v : graph[u]) if(v!=p[u])
            d[v]=d[u]+1, p[v]=u, s += ss=dfs(v), best = max(best,{ss,v});
        h[u] = best.second; return s;
    }
    int lca(int u, int v){
        for(; r[u]!=r[v]; v=p[r[v]]) if(d[r[u]] > d[r[v]]) swap(u,v);
        return d[u] < d[v] ? u : v;
    }
};
```

3.16 HLD with Segtree

Complexity: $O(n \lg^2 n)$

```
#include "../datastructures/segmenttree.cpp"
template <class T, T(*op)(T, T), T ident>
struct HLD { //graph may contain childs only
    int V; vvi &graph; SegmentTree<T,op,ident> st;
    vi p, r, d, h, t; // parents, path-root, depth heavy, tree index
    HLD(vvi &graph, vector<T> &init, int root = 0) :
        V(graph.size()), graph(graph), st({}),
    p(V,-1), r(V,-1), d(V,0), h(V,-1), t(V,-1){
        dfs(root); int k=0; vector<T> v(V);
```

```
        for(int i=0; i<V; ++i) if (p[i]==-1 || h[p[i]]!=i)
            for (int j=i; j!=-1; j=h[j]) r[j] = i, v[k]=init[j], t[j]=k++;
        st={v};
    }
    int dfs(int u){
        ii best={-1,-1}; int s=1, ss;   // best, size (of subtree)
        for(auto &v : graph[u]) if(v!=p[u])
            d[v]=d[u]+1, p[v]=u, s += ss=dfs(v), best = max(best,{ss,v});
        h[u] = best.second; return s;
    }
    int lca(int u, int v){
        for(; r[u]!=r[v]; v=p[r[v]]) if(d[r[u]] > d[r[v]]) swap(u,v);
        return d[u] < d[v] ? u : v;
    }
    void update(int u, ll v){ st.update(t[u],v); }
    T query(int u, int v){
        T a = ident;
        for(; r[u]!=r[v]; v=p[r[v]]){
            if(d[r[u]] > d[r[v]]) swap(u,v);
            a = op(a,st.query(t[r[v]], t[v]));
        }
        if(d[u] > d[v]) swap(u,v);
        return op(a,st.query(t[u],t[v])); // t[u]+1 if data is on edges
    }
};
```

4 Flow and Matching

4.1 Flow Graph

Structure used by the following flow algorithms.

```
using F = ll; using W = ll; // types for flow and weight/cost
struct S{
    const int v;            // neighbour
    const int r;    // index of the reverse edge
    F f;            // current flow
    const F cap;    // capacity
    const W cost;   // unit cost
    S(int v, int ri, F c, W cost = 0) :
        v(v), r(ri), f(0), cap(c), cost(cost) {}
};
struct FlowGraph : vector<vector<S>> {
    FlowGraph(size_t n) : vector<vector<S>>(n) {}
    void add_edge(int u, int v, F c, W cost = 0){ auto &t = *this;
        t[u].emplace_back(v, t[v].size(), c, cost);
        t[v].emplace_back(u, t[u].size()-1, 0, -cost);
    }
};
```

4.2 Dinic

Complexity: $O(V^2E)$ **Dependencies:** Flow Graph

```
#include "flowgraph.cpp"
struct Dinic{
    FlowGraph &edges; int V,s,t;
    vi l; vector<vector<S>::iterator> its; // levels and iterators
    Dinic(FlowGraph &edges, int s, int t) :
        edges(edges), V(edges.size()), s(s), t(t), l(V,-1), its(V) {}
    ll augment(int u, F c) { // we reuse the same iterators
        if (u == t) return c;
        for(auto &i = its[u]; i != edges[u].end(); i++){
            auto &e = *i;
            if (e.cap > e.f && l[u] < l[e.v]) {
                auto d = augment(e.v, min(c, e.cap - e.f));
                if (d > 0) { e.f += d; edges[e.v][e.r].f -= d; return d; }
            }   }
        return 0;
    }
    ll run() {
        ll flow = 0, f;
        while(true) {
            fill(l.begin(), l.end(),-1); l[s]=0; // recalculate the layers
            queue<int> q; q.push(s);
            while(!q.empty()){
                auto u = q.front(); q.pop();
                for(auto &&e : edges[u]) if(e.cap > e.f && l[e.v]<0)
                    l[e.v] = l[u]+1, q.push(e.v);
            }
            if (l[t] < 0) return flow;
            for (int u = 0; u < V; ++u) its[u] = edges[u].begin();
            while ((f = augment(s, INF)) > 0) flow += f;
        }   }
};
```

4.3 Minimum Cut Inference

The maximum flow equals the minimum cut. Only use this if the specific edges are needed. Run a flow algorithm in advance. **Complexity:** $O(V+E)$ **Dependencies:** Flow Network

```
void imc_dfs(FlowGraph &fg, int u, vb &cut) {
    cut[u] = true;
    for (auto &&e : fg[u]) {
        if (e.cap > e.f && !cut[e.v])
            imc_dfs(fg, e.v, cut);
    }
}
ll infer_minimum_cut(FlowGraph &fg, int s, vb &cut) {
    cut.assign(fg.size(), false);
    imc_dfs(fg, s, cut);
    ll cut_value = 0LL;
    for (size_t u = 0; u < fg.size(); ++u) {
        if (!cut[u]) continue;
        for (auto &&e : fg[u]) {
            if (cut[e.v]) continue;
            cut_value += e.cap;
            // The edge e from u to e.v is
            // in the minimum cut.
        }
    }
    return cut_value;
}
```

4.4 Min cost flow

Dependencies: Flow Graph

```
#include "flowgraph.cpp"
using F = ll; using W = ll; W WINF = LLINF; F FINF = LLINF;
struct Q{ int u; F c; W w;}; // target, maxflow and total cost/weight
bool operator>(const Q &l, const Q &r){return l.w > r.w;}
struct Edmonds_Karp_Dijkstra{
    FlowGraph &g; int V,s,t; vector<W> pot;
    Edmonds_Karp_Dijkstra(FlowGraph &g, int s, int t) :
        g(g), V(g.size()), s(s), t(t), pot(V) {}
    pair<F,W> run() { // return pair<f, cost>
        F maxflow = 0; W cost = 0;          // Bellmann-Ford for potentials
        fill(pot.begin(),pot.end(),WINF); pot[s]=0;
        for (int i = 0; i < V - 1; ++i) {
            bool relax = false;
            for (int u = 0; u < V; ++u) if(pot[u] != WINF) for(auto &e : g[u])
                if(e.cap>e.f)
                    if(pot[u] + e.cost < pot[e.v])
                        pot[e.v] = pot[u] + e.cost, relax=true;
            if(!relax) break;
        }
        for (int u = 0; u < V; ++u) if(pot[u] == WINF) pot[u] = 0;
        while(true){
            priority_queue<Q,vector<Q>,greater<Q>> q;
            vector<vector<S>::iterator> p(V,g.front().end());
            vector<W> dist(V, WINF); F f, tf = -1;
            q.push({s, FINF, 0}); dist[s]=0;
            while(!q.empty()){
                int u = q.top().u; W w = q.top().w;
                f = q.top().c; q.pop();
                if(w!=dist[u]) continue; if(u==t && tf < 0) tf = f;
                for(auto it = g[u].begin(); it!=g[u].end(); it++){
                    auto &e = *it;
                    W d =  w + e.cost + pot[u] - pot[e.v];
                    if(e.cap>e.f && d < dist[e.v]){
                        q.push({e.v, min(f, e.cap-e.f),dist[e.v] = d});
                        p[e.v]=it;
                    }   }   }
            auto it = p[t];
            if(it == g.front().end()) return {maxflow,cost};
            maxflow += f = tf;
            while(it != g.front().end()){
                auto &r = g[it->v][it->r];
                cost += f * it->cost; it->f+=f;
                r.f -= f; it = p[r.v];
            }
            for (int u = 0; u < V; ++u) if(dist[u]!=WINF) pot[u] += dist[u];
        }
    }
};
```

4.5 Min edge capacities

Make a supersource S and supersink T. When there are a lowerbound $l(u,v)$ and upperbound $c(u,v)$, add edge with capacity $c-l$. Furthermore, add (t,s) with capacity ∞.

$$M(u) = \sum_v l(v,u) - \sum_v l(u,v)$$

If $M(u) > 0$, add (S,u) with capacity $M(u)$. Otherwise add (u,T) with capacity $-M(u)$. Run Dinic to find a max flow. This is a feasible flow in the original graph if all edges from S are saturated. Run Dinic again in the residual graph of the original problem to find the maximal feasible flow.

4.6 Min vertex capacities

$x(u)$ is the amount of flow that is extracted at u, or inserted when $x(u) < 0$. If $\sum_u s(u) > 0$, add edge $(t,\tilde{t})$ with capacity ∞, and set $x(\tilde{t}) = -\sum_u x(u)$. Otherwise add $(\tilde{s},s)$ and set $x(\tilde{s}) = -\sum_u x(u)$. $\tilde{s}$ or $\tilde{t}$ is the new source/sink. Now, add S and T, (t,s) with capacity ∞. If $x(u) > 0$, add (S,u) with capacity $x(u)$. Otherwise add (u,T) with capacity $x(u)$. Use Dinic to find a max flow. If all edges from S are saturated, this is a feasible flow. Run Dinic again in the residual graph to find the maximal feasible flow.

5 Combinatorics & Probability

5.1 Stable Marriage Problem

If $m = w$, the algorithm finds a complete, optimal matching. `mpref[i][j]` gives the id of the j'th preference of the i'th man. `wpref[i][j]` gives the preference the j'th woman assigns to the i'th man. Both `mpref` and `wpref` should be zero-based permutations. **Complexity:** $O(mw)$

```
void stable_marriage(vvi &mpref, vvi &wpref, vi &mmatch) {
    size_t M = mpref.size(), W = wpref.size();
    vi wmatch(W, -1);
    mmatch.assign(M, -1);
    vector<size_t> mnext(M, 0);
    stack<size_t> st;
    for (size_t m = 0; m < M; ++m) st.push(m);

    while (!st.empty()) {
        size_t m = st.top(); st.pop();
        if (mmatch[m] != -1) continue;
        if (mnext[m] >= W) continue;

        size_t w = mpref[m][mnext[m]++];
        if (wmatch[w] == -1) {
            mmatch[m] = w;
            wmatch[w] = m;
        } else {
            size_t mp = size_t(wmatch[w]);
            if (wpref[w][m] < wpref[w][mp]) {
                mmatch[m] = w;
                wmatch[w] = m;
                mmatch[mp] = -1;
                st.push(mp);
            } else  st.push(m);
        }
    }
}
```

5.2 KP procedure

Solves a two variable single constraint integer linear programming problem. It can be extended to an arbitrary number of constraints by inductively decomposing the constrained region into its binding constraints (hence the L and U), and solving for each region. **Complexity:** $O(d^2 log(d) log(log(d)))$

```
ll solve_single(ll c, ll a, ll b, ll L, ll U) {
    if (c <= 0) return max(0LL, L);
    else return min(U, b / a);
}
ll cdiv(ll a, ll b) { return ceil(a / ll(b)); }

pair<ll, ll> KP(ll c1, ll c2, ll a1, ll a2, ll b, ll L, ll U) {
    // Trivial solutions
    if (b  <  0) return {-LLINF, -LLINF};
    if (c1 <= 0) return {L, solve_single(c2, a2, b - a1 * L, 0, LLINF)};
    if (c2 <= 0) return {solve_single(c1, a1, b, L, U), 0};
    if (a1 == 0) return {U, solve_single(c2, a2, b, 0, LLINF)};
    if (a2 == 0) return {0, LLINF};
    if (L  == U) return {L, solve_single(c2, a2, b - a1 * L, 0, LLINF) };
    if (b  == 0) return {0, 0};
    // Bound U if possible and recursively solve
    if (U != LLINF) U = min(U, b / a1);
    if (L != 0 || U != LLINF) {
        pair<ll, ll>
            kp = KP(c1, c2, a1, a2, b-cdiv(b-a1*U,a2)*a2-a1*L, 0, LLINF),
            s1 = {U, (b - a1 * U) / a2 },
            s2 = {L + kp.first, cdiv(b - a1 * U, a2) + kp.second};
        return (c1*s1.first+c2*s1.second > c1*s2.first+c2*s2.second ?s1:s2);
    } else if (a1 < a2) {
        pair<ll, ll> s = KP(c2, c1, a2, a1, b, 0, LLINF);
        return pair<ll, ll>(s.second, s.first);
    } else {
        ll k = a1 / a2, p = a1 - k * a2;
        pair<ll, ll> kp = KP(c1-c2*k, c2, p, a2, b-k*(b/a1)*a2, 0, b/a1);
        return {kp.first, kp.second - k * kp.first + k * (b/a1)};
    }
}
```

5.3 2-SAT

Complexity: $O(|\text{variables}| + |\text{implications}|)$ **Dependencies:** Tarjan's

```
#include "../graphs/tarjan.cpp"
struct TwoSAT {
    int n;
    vvi imp; // implication graph
    Tarjan tj;

    TwoSAT(int _n) : n(_n), imp(2 * _n, vi()), tj(imp) { }

    // Only copy the needed functions:
    void add_implies(int c1, bool v1, int c2, bool v2) {
        int u = 2 * c1 + (v1 ? 1 : 0),
            v = 2 * c2 + (v2 ? 1 : 0);
        imp[u].push_back(v);        //  u =>  v
        imp[v^1].push_back(u^1);    // -v => -u
    }
    void add_equivalence(int c1, bool v1, int c2, bool v2) {
        add_implies(c1, v1, c2, v2);
        add_implies(c2, v2, c1, v1);
    }
    void add_or(int c1, bool v1, int c2, bool v2) {
        add_implies(c1, !v1, c2, v2);
    }
    void add_and(int c1, bool v1, int c2, bool v2) {
        add_true(c1, v1); add_true(c2, v2);
    }
    void add_xor(int c1, bool v1, int c2, bool v2) {
        add_or(c1, v1, c2, v2);
        add_or(c1, !v1, c2, !v2);
    }
    void add_true(int c1, bool v1) {
        add_implies(c1, !v1, c1, v1);
    }

    // on true: a contains an assignment.
    // on false: no assignment exists.
    bool solve(vb &a) {
        vi com;
        tj.find_sccs(com);
        for (int i = 0; i < n; ++i)
            if (com[2 * i] == com[2 * i + 1])
                return false;

        vvi bycom(com.size());
        for (int i = 0; i < 2 * n; ++i)
            bycom[com[i]].push_back(i);

        a.assign(n, false);
        vb vis(n, false);
        for(auto &&component : bycom){
            for (int u : component) {
                if (vis[u / 2]) continue;
                vis[u / 2] = true;
                a[u / 2] = (u % 2 == 1);
            }
        }
        return true;
    }
};
```

6 Geometry

6.1 Essentials

```
using C = ld;   // could be long long or long double
constexpr C EPS = 1e-10;    // change to 0 for C=ll
struct P {      // may also be used as a 2D vector
    C x, y;
    P(C x = 0, C y = 0) : x(x), y(y) {}
    P operator+ (const P &p) const { return {x + p.x, y + p.y}; }
    P operator- (const P &p) const { return {x - p.x, y - p.y}; }
    P operator* (C c) const { return {x * c, y * c}; }
```

```
    P operator/ (C c) const { return {x / c, y / c}; }
    bool operator==(const P &r) const { return y == r.y && x == r.x; }
    C lensq() const { return x*x + y*y; }
    C len() const { return sqrt(lensq()); }
};
C sq(C x){ return x*x; }
C dot(P p1, P p2){ return p1.x*p2.x + p1.y*p2.y; }
C dist(P p1, P p2) { return (p1-p2).len(); }
C det(P p1, P p2) { return p1.x * p2.y - p1.y * p2.x; }
C det(P p1, P p2, P o) { return det(p1-o, p2-o); }
C det(vector<P> ps) {
    C sum = 0; P prev = ps.back();
    for(auto &p : ps) sum+=det(p,prev), prev=p;
    return sum;
}
C area(P p1, P p2, P p3) { return abs(det(p1, p2, p3))/C(2); }
C area(vector<P> poly) { return abs(det(poly))/C(2); }
int sign(C c){ return (c > C(0)) - (c < C(0)); }
int ccw(P p1, P p2, P p3) { return sign(det(p1, p2, p3)); }
// bool: non-parallel (P is valid), p = a*l1+(1-a)*l2 = b*r1 + (1-b)*r2
pair<bool,P> intersect(P l1, P l2, P r1, P r2, ld &a, ld &b, bool &intern){
    P dl = l2-l1, dr = r2-r1; ld d = det(dl,dr);
    if(abs(d)<=EPS) return {false,{0,0}};   // parallel
    C x = det(l1,l2)*(r1.x-r2.x) - det(r1,r2)*(l1.x-l2.x);
    C y = det(l1,l2)*(r1.y-r2.y) - det(r1,r2)*(l1.y-l2.y);
    P p = {x/d, y/d}; a = det(r1-l1,dr)/d; b = det(r1-l1,dl)/d;
    intern = 0<= a && a <= 1 && 0 <= b && b <= 1;
    return {true,p};
}
P project(P p1, P p2, P p){  // Project p on the line p1-p2
    return p1 + (p2-p1) * dot(p-p1,p2-p1)/(p2-p1).lensq(); }
P reflection(P p1, P p2, P p){ return project(p1,p2,p)*2-p; }
struct L {      // also a 3D point
    C a, b, c;  // ax + by + cz = 0
    L(C a = 0, C b = 0, C c = 0) : a(a), b(b), c(c) {}
    L(P p1, P p2) : a(p2.y-p1.y), b(p1.x-p2.x), c(p2.x*p1.y - p2.y*p1.x) {}
    void to_points(P &p1, P &p2){
        if(abs(a)<=EPS) p1 = {0, -c/b}, p2 = {1, -(c+a)/b};
        else p1 = {-c/a, 0}, p2 = {-(c+b)/a, 1};
    }
};
L cross(L p1, L p2){
    return {p1.b*p2.c-p1.c*p2.b, p1.c*p2.a-p1.a*p2.c, p1.a*p2.b-p1.b*p2.a};
}
pair<bool,P> intersect(L l1, L l2) {
    L p = cross(l1,l2);
    return {p.c!=0, {p.a/p.c, p.b/p.c}};
}

struct Circle{ P p; C r; };
vector<P> intersect(const Circle& cc, const L& l){
    const double &x = cc.p.x, &y = cc.p.y, &r = cc.r, &a=l.a,&b=l.b,&c=l.c;
    double n = a*a + b*b, t1 = c + a*x + b*y, D = n*r*r  - t1*t1;
    if(D<0) return {};
    double xmid = b*b*x - a*(c + b*y), ymid = a*a*y - b*(c + a*x);
    if(D==0) return {P{xmid/n, ymid/(n)}};
    double sd = sqrt(D);
    return {P{(xmid - b*sd)/n,(ymid + a*sd)/n},
            P{(xmid + b*sd)/n,(ymid - a*sd)/n}};
}
vector<P> intersect(const Circle& c1, const Circle& c2){
    C x = c1.p.x-c2.p.x, y = c1.p.y-c2.p.y;
    const C &r1 = c1.r, &r2 = c2.r;
    C n = x*x+y*y, D = -(n - (r1+r2)*(r1+r2))*(n - (r1-r2)*(r1-r2));
    if(D<0) return {};
    C xmid = x*(-r1*r1+r2*r2+n), ymid = y*(-r1*r1+r2*r2+n);
    if(D==0) return {P{c2.p.x + xmid/(2.*n),c2.p.y + ymid/(2.*n)}};
    double sd = sqrt(D);
    return {P{c2.p.x + (xmid - y*sd)/(2.*n),c2.p.y + (ymid + x*sd)/(2.*n)},
            P{c2.p.x + (xmid + y*sd)/(2.*n),c2.p.y + (ymid - x*sd)/(2.*n)}};
}
```

6.2 Convex Hull

Complexity: $O(n \log n)$ **Dependencies:** Geometry Essentials

```
struct point { ll x, y; };
bool operator==(const point &l, const point &r) {
    return l.x == r.x && l.y == r.y; }

ll dsq(const point &p1, const point &p2) {
    return (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y); }
ll det(ll x1, ll y1, ll x2, ll y2) {
    return x1 * y2 - x2 * y1; }
ll det(const point &p1, const point &p2, const point &d) {
    return det(p1.x - d.x, p1.y - d.y, p2.x - d.x, p2.y - d.y); }
bool comp_lexo(const point &l, const point &r) {
    return l.y != r.y ? l.y < r.y : l.x < r.x; }
bool comp_angl(const point &l, const point &r, const point &c) {
    ll d = det(l, r, c);
    if (d != 0) return d > 0;
    else return dsq(c, l) < dsq(c, r);
}

struct ConvexHull {
    vector<point> &p;
    vector<int> h;  // incides of the hull in p, ccw
    ConvexHull(vector<point> &_p) : p(_p) { compute_hull(); }
    void compute_hull() {
        int pivot = 0, n = p.size();
        vector<int> ps(n + 1, 0);
        for (int i = 1; i < n; ++i) {
            ps[i] = i;
            if (comp_lexo(p[i], p[pivot])) pivot = i;
        }
        ps[0] = ps[n] = pivot; ps[pivot] = 0;
        sort(ps.begin()+1, ps.end()-1, [this, &pivot](int l, int r) {
            return comp_angl(p[l], p[r], p[pivot]); });

        h.push_back(ps[0]);
        size_t i = 1; ll d;
        while (i < ps.size()) {
            if (p[ps[i]] == p[h.back()]) { i++; continue; }
            if (h.size() < 2 || ((d = det(p[h.end()[-2]],
                p[h.back()], p[ps[i]])) > 0)) { // >= for col.
                h.push_back(ps[i]);
```

```
                i++; continue;
            }
            if (p[h.end()[-2]] == p[ps[i]]) { i++; continue; }
            h.pop_back();
            if (d == 0) h.push_back(ps[i]);
        }
        if (h.size() > 1 && h.back() == pivot) h.pop_back();
    }
};

// Note: if h.size() is small (<5), consider brute forcing to avoid
// the usual nasty computational-geometry-edge-cases.
void rotating_calipers(vector<point> &p, vector<int> &h) {
    int n = h.size(), i = 0, j = 1, a = 1, b = 2;
    while (i < n) {
        if (det(p[h[j]].x - p[h[i]].x, p[h[j]].y - p[h[i]].y,
            p[h[b]].x - p[h[a]].x, p[h[b]].y - p[h[a]].y) >= 0) {
            a = (a + 1) % n;
            b = (b + 1) % n;
        } else {
            ++i; // NOT %n!!
            j = (j + 1) % n;
        }
        // Make computations on the pairs: h[i%n], h[a] and h[j], h[a]
    }
}
```

6.3 Upper envelope

To find the envelope of lines $a_i + b_i x$, find the convex hull of points (b_i, a_i). Add $(0, -\infty)$ for upper envelope, and $(0, +\infty)$ for lower envelope.

6.4 Formulae

$$[ABC] = rs = \frac{1}{2}ab\sin\gamma = \frac{abc}{4R} = \sqrt{s(s-a)(s-b)(s-c)} = \frac{1}{2}\left|(B-A, C-A)^T\right|$$

$$s = \frac{a+b+c}{2} \qquad 2R = \frac{a}{\sin\alpha}$$

cosine rule: $c^2 = a^2 + b^2 - 2ab\cos\gamma$

Euler: $1 + CC = V - E + F$

Pick: $\text{Area} = \text{interior points} + \frac{\text{boundary points}}{2} - 1$

$$p \cdot q = |p||q|\cos(\theta) \qquad |p \times q| = |p||q|\sin(\theta)$$

Rotatie $(x'; y') = (\cos(\theta), -\sin(\theta); \sin(\theta), \cos(\theta))\,(x; y)$

Projectie x op y $p(x,y) = \frac{x \cdot y}{y \cdot y} y$

Given a non-self-intersecting closed polygon on n vertices, given as (x_i, y_i), its centroid (C_x, C_y) is given as:

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \quad C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) = \text{polygon area}$$

7 Mathematics

7.1 Primes

$$10^3 + \{-9, -3, 9, 13\}, \quad 10^6 + \{-17, 3, 33\}, \quad 10^9 + \{7, 9, 21, 33, 87\}$$

```
#include "numbertheory.cpp"
ll SIZE; vector<bool> bs; vector<ll> primes, mf;// mf[i]==i when prime

void sieve(ll size = 1e6) { // call at start in main!
    SIZE = size; bs.assign(SIZE+1,1);
    bs[0] = bs[1] = 0;
    for (ll i = 2; i <= SIZE; i++) if (bs[i]) {
        for (ll j = i * i; j <= SIZE; j += i) bs[j] = 0;
        primes.push_back(i);
    }
}
bool is_prime(ll n) { // for N <= SIZE^2
    if (n <= SIZE) return bs[n];
    for(const auto &prime : primes)
        if (n % prime == 0) return false;
    return true;
}
struct Factor{ll p; ll exp;}; using FS = vector<Factor>;
FS factor(ll n) { FS fs;
    for(const auto &p: primes){ ll exp=0;
        if(n==1 || p*p > n) break;
        while(n % p == 0) n/=p, exp++;
        if(exp>0) fs.push_back({p,exp});
    }
    if (n != 1) fs.push_back({n,1});
    return fs;
}

void sieve2(ll size=1e6) {  // call at start in main!
    SIZE = size; mf.assign(SIZE+1,-1);
    mf[0] = mf[1] = 1;
    for (ll i = 2; i <= SIZE; i++) if (mf[i] < 0) {
        mf[i] = i;
        for (ll j = i * i; j <= SIZE; j += i)
            if(mf[j] < 0) mf[j] = i;
        primes.push_back(i);
    }
}
bool is_prime2(ll n) { assert(n<=SIZE); return mf[n]==n; }
FS factor2(ll n){ FS fs;
    for(; n>1; n/=mf[n])
        if(!fs.empty() && fs.back().p== mf[n]) fs.back().exp++;
        else fs.push_back({mf[n],1});
    return fs;
}

vector<ll> divisors(const FS &fs){ vector<ll> ds{1};
```

```
    ll s=1; for(auto &f:fs) s*=f.exp+1; ds.reserve(s);
    for(auto f : fs) for(auto d : ds) for(ll i=0; i<f.exp; ++i)
        ds.push_back(d*=f.p);
    return ds;
}
ll num_div(  const FS &fs) { ll d = 1;
    for(auto &f : fs) d *= f.exp+1; return d; }
ll sum_div(  const FS &fs) { ll s = 1;
    for(auto &f : fs) s *= (pow(f.p,f.exp+1)-1)/(f.p-1); return s; }
ll phi(ll n, const FS &fs) { ll p = n;
    for(auto &f : fs) p -= p/f.p; return p; }
ll ord(ll n, ll m, const FS &fs){ ll o = phi(m,fs); // n^ord(n,m)=1 mod m
    for(auto f : factor(o)) while(f.exp-- && powmod(n,o/f.p,m)==1) o/=f.p;
    return o; }
```

7.2 Euler Phi

Complexity: $O(n \log\log n)$

```
void calculate_phi(int n, vector<ll> &phi) {
    phi.resize(n);
    iota(phi.begin(), phi.end(), 0); // numeric
    for (ll i=2; i<=n; ++i) if (phi[i] == i)
        for (ll j=i; j<=n; j+=i) phi[j] -= phi[j]/i;
}
```

7.3 Number theoretic algorithms

```
ll gcd(ll a, ll b) { while (b) { a %= b; swap(a, b); } return a; }
ll lcm(ll a, ll b) { return (a / gcd(a, b)) * b;                 }
ll mod(ll a, ll b) { return ((a % b) + b) % b;                 }

// Finds x, y s.t. ax + by = d = gcd(a, b).
void extended_euclid(ll a, ll b, ll &x, ll &y, ll &d) {
    ll xx = y = 0;
    ll yy = x = 1;
    while (b) {
        ll q = a / b;
        ll t = b; b = a % b; a = t;
        t = xx; xx = x - q * xx; x = t;
        t = yy; yy = y - q * yy; y = t;
    }
    d = a;
}

// solves ab = 1 (mod n), -1 on failure
ll mod_inverse(ll a, ll n) {
    ll x, y, d;
    extended_euclid(a, n, x, y, d);
    return (d > 1 ? -1 : mod(x, n));
}

// (a*b)%m
ll mulmod(ll a, ll b, ll m){
    ll x = 0, y=a%m;
    while(b>0){
        if(b&1)
            x = (x+y)%m;
        y = (2*y)%m;
        b/=2;
    }
    return x % m;
}
ll mulmod2(ll a, ll b, ll m){ return __int128(a)*b%m; }

ll pow(ll b, ll e) {        // b^e in logarithmic time
    ll p = e<2 ? 1 : pow(b*b,e/2);
    return e&1 ? p*b : p;
}

// Finds b^e % m in O(lg n) time, ensure that b < m to avoid overflow!
ll powmod(ll b, ll e, ll m) {
    ll p = e<2 ? 1 : powmod((b*b)%m,e/2,m);
    return e&1 ? p*b%m : p;
}

// Solve ax + by = c, returns false on failure.
bool linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
    ll d = gcd(a, b);
    if (c % d) {
        return false;
    } else {
        x = c / d * mod_inverse(a / d, b / d);
        y = (c - a * x) / b;
        return true;
    }
}

ll binom(ll n, ll k){
    ll ans = 1;
    for(ll i = 1; i <= min(k,n-k); ++i) ans = ans*(n+1-i)/i;
    return ans;
}

// Solves x = a1 mod m1, x = a2 mod m2, x is unique modulo lcm(m1, m2).
// Returns {0, -1} on failure, {x, lcm(m1, m2)} otherwise.
pair<ll, ll> crt(ll a1, ll m1, ll a2, ll m2) {
    ll s, t, d;
    extended_euclid(m1, m2, s, t, d);
    if (a1 % d != a2 % d) return {0, -1};
    return {mod(s * a2 * m1 + t * a1 * m2, m1 * m2) / d, m1 / d * m2};
}

// Solves x = ai mod mi. x is unique modulo lcm mi.
// Returns {0, -1} on failure, {x, lcm mi} otherwise.
pair<ll, ll> crt(vector<ll> &a, vector<ll> &m) {
    pair<ll, ll> res = {a[0], m[0]};
    for (ull i = 1; i < a.size(); ++i) {
        res = crt(res.first, res.second, mod(a[i], m[i]), m[i]);
        if (res.second == -1) break;
    }
    return res;
}
```

7.4 Lucas' theorem

```
#include "./primes.cpp"
ll lucas(ll n, ll k, ll p){ // calculate (n \choose k) % p
    ll ans = 1;
    while(n){
        ll np = n%p, kp = k%p;
        if(kp > np) return 0;
        ans *= binom(np,kp);
        n /= p; k /= p;
    }
    return ans;
}
```

7.5 Finite Field

```
#include "./numbertheory.cpp"
template<ll p,ll w> // prime, primitive root
struct Field { using T = Field; ll x; Field(ll x=0) : x{x} {}
    T operator+(T r) const { return {(x+r.x)%p}; }
    T operator-(T r) const { return {(x-r.x+p)%p}; }
    T operator*(T r) const { return {(x*r.x)%p}; }
    T inv(){ return {mod_inverse(x,p)}; }
    static T root(ll k) { assert( (p-1)%k==0 );     // (p-1)%k == 0?
        auto r = powmod(w,(p-1)/abs(k),p);          // k-th root of unity
        return k>=0 ? T{r} : T{r}.inv();
    }
};
using F1 = Field<1004535809,3 >;
using F2 = Field<1107296257,10>; // 1<<30 + 1<<25 + 1
using F3 = Field<2281701377,3 >; // 1<<31 + 1<<27 + 1
```

7.6 Complex Numbers

Faster-than-built-in complex numbers

```
constexpr ld pi = 3.1415926535897932384626433;
struct Complex { using T = Complex; ld u,v;
    Complex(ld u=0, ld v=0) : u{u}, v{v} {}
    T operator+(T r) const { return {u+r.u, v+r.v}; }
    T operator-(T r) const { return {u-r.u, v-r.v}; }
    T operator*(T r) const { return {u*r.u - v*r.v, u*r.v + v*r.u}; }
    T operator/(T r) {
        auto norm = r.u*r.u+r.v*r.v;
        return {(u*r.u + v*r.v)/norm, (v*r.u - u*r.v)/norm};
    }
    T inv(){ return T{1,0}/ *this; }
    static T root(ll k){ return {cos(2*pi/k), sin(2*pi/k)}; }
};
```

7.7 Fast Fourier Transform

Calculates the discrete convolution of two vectors. Note that the method accepts and outputs complex numbers, and the input is changed in place. **Complexity:** $O(n \log n)$
Dependencies: Bitmasking, Complex Numbers

```
#include "../helpers/bitmasking.cpp"
#include "./complex.cpp"
#include "./field.cpp"
using T = Complex;  // using T=F1,F2,F3
void fft(vector<T> &A, int p, bool inv = false) {
    int N = 1<<p;
    for(int i = 0, r = 0; i < N; ++i, r = brinc(r, p))
        if (i < r) swap(A[i], A[r]);
    for (int m = 2; m <= N; m <<= 1) {
        T w, w_m = T::root(inv ? -m : m);
        for (int k = 0; k < N; k += m) {
            w = T{1};
            for (int j = 0; j < m/2; ++j) {
                T t = w * A[k + j + m/2];
                A[k + j + m/2] = A[k + j] - t;
                A[k + j] = A[k + j] + t;
                w = w * w_m;
            }
        }
    }
    if(inv){ T inverse = T(N).inv(); for(auto &x : A) x = x*inverse; }
}
// convolution leaves A and B in frequency domain state
// C may be equal to A or B for in-place convolution
void convolution(vector<T> &A, vector<T> &B, vector<T> &C){
    int s = A.size() + B.size() - 1;
    int q = 32 - __builtin_clz(s-1), N=1<<q;    // fails if s=1
    A.resize(N,{}); B.resize(N,{}); C.resize(N,{});
    fft(A, q, false); fft(B, q, false);
    for (int i = 0; i < N; ++i) C[i] = A[i] * B[i];
    fft(C, q, true); C.resize(s);
}
void convolution(vector<vector<T>> &ps, vector<T> &C){
    int s=1; for(auto &p : ps) s+=p.size()-1;
    int q = 32 - __builtin_clz(s-1), N=1<<q;    // fails if s=1
    C.assign(N,{1});
    for(auto &p : ps){ p.resize(N,{}); fft(p, q, false);
        for(int i = 0; i < N; ++i) C[i] = C[i] * p[i];
    }
    fft(C, q, true); C.resize(s);
}
void square_inplace(vector<T> &A) {
    int s = 2*A.size()-1, q = 32 - __builtin_clz(s-1), N=1<<q;
    A.resize(N,{}); fft(A, q, false);
    for(auto &x : A) x = x*x;
    fft(A, q, true); A.resize(s);
}
```

7.8 Matrix equation solver

Solve $MX = A$ for X, and write the square matrix M in reduced row echelon form, where each row starts with a 1, and this 1 is the only nonzero value in its column.

```
using T = double;
constexpr T EPS = 1e-8;
template<int R, int C>
using M = array<array<T,C>,R>;  // matrix
```

```
template<int R, int C>
T ReducedRowEchelonForm(M<R,C> &m, int rows) {  // return the determinant
    int r = 0; T det = 1;                          // MODIFIES the input
    for(int c = 0; c < rows && r < rows; c++) {
        int p = r;
        for(int i=r+1; i<rows; i++) if(abs(m[i][c]) > abs(m[p][c])) p=i;
        if(abs(m[p][c]) < EPS){ det = 0; continue; }
        swap(m[p], m[r]);       det *= ( (p-r)%2 ? -1 : 1 );
        T s = 1.0 / m[r][c], t; det *= m[r][c];
        REP(j,C) m[r][j] *= s;               // make leading term in row 1
        REP(i,rows) if (i!=r){ t = m[i][c]; REP(j,C) m[i][j] -= t*m[r][j]; }
        ++r;
    }
    return det;
}
bool error, inconst;    // error => multiple or inconsistent
template<int R,int C>   // Mx = a; M:R*R, v:R*C => x:R*C
M<R,C> solve(const M<R,R> &m, const M<R,C> &a, int rows){
    M<R,R+C> q;
    REP(r,rows){
        REP(c,rows) q[r][c] = m[r][c];
        REP(c,C) q[r][R+c] = a[r][c];
    }
    ReducedRowEchelonForm<R,R+C>(q,rows);
    M<R,C> sol; error = false, inconst = false;
    REP(c,C) for(auto j = rows-1; j >= 0; --j){
        T t=0; bool allzero=true;
        for(auto k = j+1; k < rows; ++k)
            t += q[j][k]*sol[k][c], allzero &= abs(q[j][k]) < EPS;
        if(abs(q[j][j]) < EPS)
            error = true, inconst |= allzero && abs(q[j][R+c]) > EPS;
        else sol[j][c] = (q[j][R+c] - t) / q[j][j];
    }
    return sol;
}
```

7.9 Matrix Exponentation

Matrix exponentation in logarithmic time.

```
#define ITERATE_MATRIX(w) for (int r = 0; r < (w); ++r) \
                          for (int c = 0; c < (w); ++c)
template <class T, int N>
struct M {
    array<array<T,N>,N> m;
    M() { ITERATE_MATRIX(N) m[r][c] = 0; }
    static M id() {
        M I; for (int i = 0; i < N; ++i) I.m[i][i] = 1; return I;
    }
    M operator*(const M &rhs) const {
        M out;
        ITERATE_MATRIX(N) for (int i = 0; i < N; ++i)
            out.m[r][c] += m[r][i] * rhs.m[i][c];
        return out;
    }
    M raise(ll n) const {
        if(n == 0) return id();
        if(n == 1) return *this;
        auto r = (*this**this).raise(n / 2);
        return (n%2 ? *this*r : r);
    }
};
```

7.10 Simplex algorithm

Maximize $c^t x$ subject to $Ax \leq b$ and $x \geq 0$. $A[m \times n]$, $b[m]$, $c[n]$, $x[n]$. Solution in x.

```
using T = long double; using vd = vector<T>; using vvd = vector<vd>;
const T EPS = 1e-9;
struct LPSolver {
    int m, n; vi B, N; vvd D;
    LPSolver(const vvd &A, const vd &b, const vd &c) :
        m(b.size()), n(c.size()), B(m), N(n+1), D(m+2, vd(n+2)) {
            REP(i,m) REP(j,n) D[i][j] = A[i][j];
            REP(i,m) B[i] = n+i, D[i][n] = -1, D[i][n+1] = b[i];
            REP(j,n) N[j] = j, D[m][j] = -c[j];
            N[n] = -1; D[m+1][n] = 1;
        }
    void Pivot(int r, int s) {
        REP(i,m+2) if (i != r) REP(j,n+2) if (j != s)
                D[i][j] -= D[r][j] * D[i][s] / D[r][s];
        REP(j,n+2) if (j != s) D[r][j] /= D[r][s];
        REP(i,m+2) if (i != r) D[i][s] /= -D[r][s];
        D[r][s] = 1.0 / D[r][s];
        swap(B[r], N[s]);
    }
    bool Simplex(int phase) {
        int x = phase == 1 ? m+1 : m;
        while (true) {
            int s = -1;
            REP(j,n+1){
                if (phase == 2 && N[j] == -1) continue;
                if (s == -1 || D[x][j] < D[x][s] ||
                    (D[x][j] == D[x][s] && N[j] < N[s])) s = j;
            }
            if (D[x][s] >= -EPS) return true;
            int r = -1;
            REP(i,m){
                if (D[i][s] <= 0) continue;
                if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
                    (D[i][n+1]/D[i][s] == D[r][n+1]/D[r][s] && B[i]< B[r]))
                    r = i;
            }
            if (r == -1) return false;
            Pivot(r, s);
        }
    }
    T Solve(vd &x) {
        int r = 0;
        for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] <= -EPS) {
            Pivot(r, n);
            if (!Simplex(1) || D[m+1][n+1] < -EPS) return -INF;
            REP(i,m) if (B[i] == -1) {
                int s = -1;
                REP(j,n+1)
```

```
                if (s == -1 || D[i][j] < D[i][s] ||
                    (D[i][j] == D[i][s] && N[j] < N[s])) s = j;
            Pivot(i, s);
        }
    }
    if (!Simplex(2)) return INF;
    x = vd(n);
    REP(i,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};
```

7.11 Game theory

A game can be reduced to Nim if it is a finite impartial game, then for any state x, $g(x) = \inf(\mathbb{N}_0 - \{g(y) : y \in F(x)\})$. Nim and its variants include:

Nim Let $X = \bigoplus_{i=1}^n x_i$, then $(x_i)_{i=1}^n$ is a winning position iff $X \neq 0$. Find a move by picking k such that $x_k > x_k \oplus X$.

Misère Nim Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles.

Staricase Nim Stones are moved down a staircase and only removed from the last pile. $(x_i)_{i=1}^n$ is an L-position if $(x_{2i-1})_{i=1}^{n/2}$ is (i.e. only look at odd-numbered piles).

Moore's Nim$_k$ The player may remove from at most k piles (Nim = Nim$_1$). Expand the piles in base 2, do a carry-less addition in base $k+1$ (i.e. the number of ones in each column should be divisible by $k+1$).

Dim$^+$ The number of removed stones must be a divisor of the pile size. The Sprague-Grundy function is $k+1$ where 2^k is the largest power of 2 dividing the pile size.

Aliquot game Same as above, except the divisor should be proper (hence 1 is also a terminal state, but watch out for size 0 piles). Now the Sprague-Grundy function is just k.

Nim (at most half) Write $n+1 = 2^m y$ with m maximal, then the Sprague-Grundy function of n is $(y-1)/2$.

Lasker's Nim Players may alternatively split a pile into two new non-empty piles. $g(4k+1) = 4k+1$, $g(4k+2) = 4k+2$, $g(4k+3) = 4k+4$, $g(4k+4) = 4k+3$ $(k \geq 0)$.

Hackenbush on trees A tree with stalks $(x_i)_{i=1}^n$ may be replaced with a single stalk with length $\bigoplus_{i=1}^n x_i$.

A useful identity: $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a\%4]$.

7.12 Formulae

Lucas $$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \mod p$$

Lagrange $$L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0 \le m \le k \\ m \neq j}} \frac{x - x_m}{x_j - x_m}$$

8 Strings

8.1 Knuth Morris Pratt

Complexity: $O(n+m)$

```
void compute_prefix_function(string &w, vi &pi) {
    pi.assign(w.length(), 0);
    int k = pi[0] = -1;

    for (int i = 1; i < w.length(); ++i) {
        while (k >= 0 && w[k + 1] != w[i])
            k = pi[k];
        if (w[k + 1] == w[i]) k++;
        pi[i] = k;
    }
}

void knuth_morris_pratt(string &s, string &w) {
    int q = -1; vi pi;
    compute_prefix_function(w, pi);
    for (int i = 0; i < s.length(); ++i) {
         while (q >= 0 && w[q + 1] != s[i]) q = pi[q];
         if (w[q + 1] == s[i]) q++;
         if (q + 1 == w.length()) {
              // Match at position (i - w.length() + 1)
              q = pi[q];
         }
    }
}
```

8.2 Z-algorithm

To match pattern P on string S: pick Φ s.t. $\Phi \notin P$, find Z of $P\Phi S$. **Complexity:** $O(n)$

```
void Z_algorithm(string &s, vector<int> &Z) {
    Z.assign(s.length(), -1);
    int L = 0, R = 0, n = s.length();
    for (int i = 1; i < n; ++i) {
        if (i > R) {
            L = R = i;
            while (R < n && s[R - L] == s[R]) R++;
            Z[i] = R - L; R--;
        } else if (Z[i - L] >= R - i + 1) {
            L = i;
            while (R < n && s[R - L] == s[R]) R++;
            Z[i] = R - L; R--;
        } else Z[i] = Z[i - L];
```

```
    }
    Z[0] = n;
}
```

8.3 Aho-Corasick

Constructs a Finite State Automaton that can match k patterns of total length m on a string of size n. **Complexity:** $O(n+m+k)$

```
template <int ALPHABET_SIZE, int (*mp)(char)>
struct AC_FSM {
    struct Node {
        int child[ALPHABET_SIZE], failure = 0, match_par = -1;
        vi match;
        Node() { for (int i = 0; i < ALPHABET_SIZE; ++i) child[i] = -1; }
    };
    vector<Node> a;
    vector<string> &words;
    AC_FSM(vector<string> &words) : words(words) {
        a.push_back(Node());
        construct_automaton();
    }
    void construct_automaton() {
        for (int w = 0, n = 0; w < words.size(); ++w, n = 0) {
            for (int i = 0; i < words[w].size(); ++i) {
                if (a[n].child[mp(words[w][i])] == -1) {
                    a[n].child[mp(words[w][i])] = a.size();
                    a.push_back(Node());
                }
                n = a[n].child[mp(words[w][i])];
            }
            a[n].match.push_back(w);
        }

        queue<int> q;
        for (int k = 0; k < ALPHABET_SIZE; ++k) {
            if (a[0].child[k] == -1) a[0].child[k] = 0;
            else if (a[0].child[k] > 0) {
                a[a[0].child[k]].failure = 0;
                q.push(a[0].child[k]);
            }
        }
        while (!q.empty()) {
            int r = q.front(); q.pop();
            for (int k = 0, arck; k < ALPHABET_SIZE; ++k) {
                if ((arck = a[r].child[k]) != -1) {
                    q.push(arck);
                    int v = a[r].failure;
                    while (a[v].child[k] == -1) v = a[v].failure;
                    a[arck].failure = a[v].child[k];
                    a[arck].match_par = a[v].child[k];
                    while (a[arck].match_par != -1 && a[a[arck].match_par].
                        match.empty())
                        a[arck].match_par = a[a[arck].match_par].match_par;
                }
            }
        }
    }

    void aho_corasick(string &sentence, vvi &matches){
        matches.assign(words.size(), vi());
        int state = 0, ss = 0;
        for (int i = 0; i < sentence.length(); ++i, ss = state) {
            while (a[ss].child[mp(sentence[i])] == -1)
                ss = a[ss].failure;
            state = a[state].child[mp(sentence[i])]
                  = a[ss].child[mp(sentence[i])];
            for (ss = state; ss != -1; ss = a[ss].match_par)
                for (int w : a[ss].match)
                    matches[w].push_back(i + 1 - words[w].length());
        }
    }
};
```

8.4 Manacher's Algorithm

Finds the largest palindrome centered at each position. **Complexity:** $O(|S|)$

```
void manacher(string &s, vector<int> &pal) {
    int n = s.length(), i = 1, l, r;
    pal.assign(2 * n + 1, 0);
    while (i < 2 * n + 1) {
        if ((i&1) && pal[i] == 0) pal[i] = 1;
        l = i / 2 - pal[i] / 2; r = (i-1) / 2 + pal[i] / 2;

        while (l - 1 >= 0 && r + 1 < n && s[l - 1] == s[r + 1])
            --l, ++r, pal[i] += 2;

        for (l = i - 1, r = i + 1; l >= 0 && r < 2 * n + 1; --l, ++r) {
            if (l <= i - pal[i]) break;
            if (l / 2 - pal[l] / 2 > i / 2 - pal[i] / 2)
                pal[r] = pal[l];
            else {  if (l >= 0)
                    pal[r] = min(pal[l], i + pal[i] - r);
                break;
            }
        }
        i = r;
}   }
```

9 DP

9.1 Convex Hull optimization

When $a_{j+1} < a_j$ and $x_{i+1} > x_i$ (otherwise sort x):

$$D_{k,i} = \min_{j<i} \{a_j \cdot x_i + D_{k-1,j}\} + c_{k,i}$$

$$D_i = \min_{j<i} \{a_j \cdot x_i + D_j\} + c_i$$

Complexity: $O(kn^2) \to O(kn)$, $O(n^2) \to O(n)$

```
#include "../geometry/essentials.cpp"   // for Point and ccw
ld eval(P p, ld x){ return x*p.x + p.y; }
// dp[k][i] = min_{j<i} (a[j]*x[i] + dp[k-1][j]=b) + c[i]
// a[j+1] < a[j], x[i+1] > x[i] (otherwise sort on x before evaluate)
```

```
// prefill dp with INF
void convex_hull_dp_2d(vi &a, vi &x, vi &b, vi &c, ll k, vi &dp){
    vector<P> v; ll n=x.size(), q=0;
    for(ll i=k-1; i<n; ++i){    // -1 only when k is 1-based
        P p(a[i-1], b[i-1]);
        while(v.size()>=2 && ccw(v[v.size()-2],v.back(),p)>0) v.pop_back();
        v.push_back(p);
        while(q+1<v.size() && eval(v[q+1],x[i]) < eval(v[q], x[i])) ++q;
        dp[i] = eval(v[q], x[i]) + c[i];
    }
}
// dp[i] = min_{j<i} (a[j]*x[i] + dp[j]) + c[i], dp[0] = c[0]
// a[j+1] < a[j], x[i+1] > x[i]
void convex_hull_dp_1d(vi &a, vi &x, vi &c, vi &dp){
    dp.assign(x.size(), 1e18); dp[0] = c[0];
    convex_hull_dp_2d(a,x,dp,c,2,dp);
}
```

9.2 Divide and Conquer

When $P_{l,r} \leq P_{l,r+1}$, solve the recursion

$$D_{k,i} = \min_{j<i} \{D_{k-1,j} + C(j,i)\}$$

Complexity: $O(kn^2) \to O(kn \lg n)$

```
// dp[k][i] = min_{j<i}{dp[k-1][j]+C[j][i]}
// when A[k][i] <= A[k][i+1]
// d:old, dp: new, calculate dp[l,r] with optimum in [optl,optr]
void compute(vi &d, vi& dp, ll l, ll r, ll optl, ll optr, ll C(ll,ll)){
    ll m = (l+r)/2; ii best{1e18, -1}; // calc dp[m]
    for(ll j=optl; j<=optr; ++j) best = min(best,{d[j]+C(j,m),j});
    dp[m] = best.first; ll opt = best.second;
    if(l<m) compute(d,dp,l,m-1,optl,opt ,C);
    if(m<r) compute(d,dp,m+1,r,opt ,optr,C);
}
vi divide_conquer_dp(vi &d, ll C(ll,ll)){
    vi dp(d.size(), 1e18);
    compute(d,dp,0,d.size()-1,0,d.size()-1, C);
    return dp;
}
```

9.3 Knuth optimization

$$D_{l,r} = \min_{l<m<r} \{D_{l,m} + D_{m,r}\} + C_{l,r} = \min_{P_{l,r-1} \leq m \leq P_{l+1,r}} \{D_{l,m} + D_{m,r}\} + C_{l,r}$$

where $P_{l,r}$ is the m for which $D_{l,r} = D_{l,m} + D_{m,r} + C_{l,r}$. Holds when $P_{l,r-1} \leq P_{l,r} \leq P_{l+1,r}$, or implied when for all $a \leq b \leq c \leq d$:

$$C_{a,c} + C_{b,d} \leq C_{a,d} + C_{b,d} \qquad C_{b,c} \leq C_{a,b}$$

Complexity: $O(n^3) \to O(n^2)$

10 Miscellaneous

10.1 LIS

Finds the longest strictly increasing subsequence. To find the longest non-decreasing subsequence, insert pairs (a_i, i). Note that the elements should be totally ordered. To find the LIS of a sequence of elements from a partially ordered set (e.g. coordinates in the plane), replace `lis[]` with a set of equivalent elements, at a cost of another $O(\log n)$ factor. **Complexity:** $O(n \log n)$

```
// Length only
template<class T>
int longest_increasing_subsequence(vector<T> &a) {
    set<T> st;
    typename set<T>::iterator it;
    for (int i = 0; i < a.size(); ++i) {
        it = st.lower_bound(a[i]);
        if (it != st.end()) st.erase(it);
        st.insert(a[i]);
    }
    return st.size();
}

// Entire sequence (indices)
template<class T>
int longest_increasing_subsequence(vector<T> &a, vector<int> &seq) {
    vector<int> lis(a.size(), 0), pre(a.size(), -1);
    int L = 0;
    for (int i = 0; i < a.size(); ++i) {
        int l = 1, r = L;
        while (l <= r) {
            int m = (l + r + 1) / 2;
            if (a[lis[m - 1]] < a[i])
                l = m + 1;
            else
                r = m - 1;
        }

        pre[i] = (l > 1 ? lis[l - 2] : -1);
        lis[l - 1] = i;
        if (l > L) L = l;
    }

    seq.assign(L, -1);
    int j = lis[L - 1];
    for (int i = L - 1; i >= 0; --i) {
        seq[i] = j;
        j = pre[j];
    }
    return L;
}
```

10.2 Randomisation

Might be useful for NP-Complete/Backtracking problems

```
#include <chrono>
using namespace chrono;
auto beg = high_resolution_clock::now();
while(high_resolution_clock::now() - beg < milliseconds(TIMELIMIT - 250)){}
```

10.3 All Nearest Smaller Values

Complexity: $O(n)$

```
void all_nearest_smaller_values(vi &a, vi &b) {
    b.assign(a.size(), -1);
    for (int i = 1; i < b.size(); ++i) {
        b[i] = i - 1;
        while (b[i] >= 0 && a[i] < a[b[i]])
            b[i] = b[b[i]];
    }
}
```

11 Helpers

11.1 Golden Section Search

For a discrete search: use binary search on the difference of successive elements, see the section on Binary Search. **Complexity:** $O(\log 1/\epsilon)$

```
#define RES_PHI (2 - ((1.0 + sqrt(5)) / 2.0))
#define EPSILON 1e-7

double gss(double (*f)(double), double leftbound, double rightbound) {
    double lb = leftbound, rb = rightbound, mlb = lb + RES_PHI * (rb - lb),
        mrb = rb + RES_PHI * (lb - rb);
    double lbv = f(lb), rbv = f(rb), mlbv = f(mlb), mrbv = f(mrb);

    while (rb - lb >= EPSILON) { // || abs(rbv - lbv) >= EPSILON) {
        if (mlbv < mrbv) { // > to maximize
            rb  = mrb;  rbv  = mrbv;
            mrb = mlb;  mrbv = mlbv;
            mlb = lb + RES_PHI * (rb - lb);
            mlbv = f(mlb);
        } else {
            lb  = mlb;  lbv  = mlbv;
            mlb = mrb;  mlbv = mrbv;
            mrb = rb + RES_PHI * (lb - rb);
            mrbv = f(mrb);
        }
    }
    return mlb; // any bound should do
}
```

11.2 Binary Search

Complexity: $O(\log n), O(\log 1/\epsilon)$

```
# define EPSILON 1e -7

// Finds the first i s.t. arr[i]>=val, assuming that arr[l] <= val <= arr[h]
int integer_binary_search(int l, int h, vector<double> &arr, double val) {
    while (l < h) {
        int m = l + (h - l) / 2;
        if (arr[m] >= val) h = m;
        else               l = m + 1;
    }
    return l;
}

// Given a monotonically increasing function f, approximately solves f(x)=c,
// assuming that f(l) <= c <= f(h)
double binary_search(double l, double h, double (*f)(double), double c) {
    while (true) {
        double m = (l + h) / 2, v = f(m);
        if (abs(v - c) < EPSILON) return m;
        if (v < c) l = m;
        else       h = m;
    }
}

// Modifying binary search to do an integer ternary search:
int integer_ternary_search(int l, int h, vector <double> &arr) {
    while (l < h) {
        int m = l + (h - l) / 2;
        if (arr[m + 1] - arr[m] >= 0) h = m;
        else l = m + 1;
    }
    return l;
}
```

11.3 Bitmasking

```
#ifdef _MSC_VER
#define popcount(x) __popcnt(x)
#else
#define popcount(x) __builtin_popcount(x)
#endif
template<typename F>    // All subsets of {0..N-1}
void iterate_subset(ll N, F f){for(ll mask=0; mask < 1ll<<N; ++mask) f(mask)
    ;}
template<typename F>    // All subsets of size k of {0..N-1}
void iterate_k_subset(ll N, ll k, F f){
    ll mask = (1ll << k) - 1;
    while (!(mask & 1ll<<N)) { f(mask);
        ll t = mask | (mask-1);
        mask = (t+1) | (((~t & -~t) - 1) >> (__builtin_ctzll(mask)+1));
    }
}
template<typename F>    // All subsets of set
void iterate_mask_subset(ll set, F f){ ll mask = set;
    do  f(mask), mask = (mask-1) & set;
    while (mask != set);
}
ll next_power_of_2(ll x) {  // used in FFT
    x  = (x - 1) | ((x - 1) >> 1);
    x |= x >> 2; x |= x >> 4; x |= x >> 8; x |= x >> 16;
    return x + 1;
}
ll brinc(ll x, ll k) {
    ll i = k - 1, s = 1 << i;
    x ^= s;
    if ((x & s) != s) {
        --i; s >>= 1;
        while (i >= 0 && ((x & s) == s))
```

```
            x = x &~ s, --i, s >>= 1;
        if (i >= 0) x |= s;
    }
    return x;
}
```

11.4 Fast IO

```
int r() {
    int sign = 1, n = 0;
    char c;
    while ((c = getchar_unlocked()) != '\n')
        switch (c) {
            case '-': sign = -1; break;
            case ' ': case '\n': return n * sign;
            default: n *= 10; n += c - '0'; break;
        }
}

void scan(ll &x){   // doesn't handle negative numbers
    char c;
    while((x=getchar_unlocked())<'0');
    for(x-='0'; '0'<=(c=getchar_unlocked()); x=10*x+c-'0');
}
void print(ll x){
    char buf[20], *p=buf;
    if(!x) putchar_unlocked('0');
    else{
        while(x) *p++='0'+x%10, x/=10;
        do putchar_unlocked(*--p); while(p!=buf);
    }
}
```

11.5 Detecting overflow

These are GNU builtins, detect both over- and underflow. Returns a boolean upon failure, otherwise the result is present in `ref`. Follow the template:
`__builtin_[u|s][add|mul|sub](ll)?_overflow(in, out, &ref)`

12 Strategies

Take a break after 2 hours.

Techniques

- Bruteforce: meet-in-the-middle, backtracking, memoization
- DP (write full draft, include ALL loop bounds), easy direction
- Precomputation
- Divide and Conquer
- Binary search
- $lg(n)$ datastructures
- Mathematical insight
- Randomisation
- Look at it backwards
- Common subproblems? Memoization
- Compute modulo primes and use CRT

WA

- Beware of typos
- Test sample input; make custom testcases
- Read carefully
- Check bounds (use long long or long double)
- EDGE CASES: $n \in \{-1, 0, 1, 2\}$. Empty list/graph?
- Off by one error (in indices or loop bounds)
- Not enough precision
- Assertions
- Missing modulo operators
- Cases that need a (completely) different approach

TLE

- Infinite loop
- Use scanf or fastIO instead of cin
- Wrong algorithm (is it theoretically fast enough)
- Micro optimizations (but probably the approach just isn't right)

RTE

- Typos
- Off by one error (in array index of loop bound)
- empty vector front/back
- return 0 at end of program