Universiteit Utrecht

Team Code Reference

Curiously Recurring

Utrechts Kampioenschap Programmeren - BAPC Preliminaries
19 september 2015

1 Templates

1.1 Vimrc

```
syntax on            colorscheme default            hi Comment ctermfg=cyan
tabstop=4 softtabstop=4 shiftwidth=4 laststatus=2 encoding=utf-8 mouse=nvc
clipboard=unnamed backspace=indent,eol,start cinoptions=:0,l1,g0,(0
noexpandtab wrap linebreak nu incsearch cindent autoindent hidden
```

1.2 C++ Template

```cpp
iostream string sstream vector list set map unordered_map queue stack bitset
tuple cstdio numeric iterator algorithm cmath chrono cassert unordered_set
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 F(v)        begin(v), end(v)
constexpr bool LOG =
#ifdef _LOG                         // -D_LOG compiler option
true;
#define _GLIBCXX_DEBUG              // for bounds checking etc
#else
false;
#endif
using ll = long long; using ii = pair<int,int>; using vi = vector<int>;
using vb = vector<bool>; using vvi = vector<vi>;
constexpr int  INF = 1e9+1; // < 1e9 - -1e9
constexpr ll LLINF = 1e18+1;
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

```
class UnionFind {
private:
    vi par, rank, size; int c;
public:
    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 union_set(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>
class FenwickTree {
private:
    vector<T> tree;
    int n;
```

```cpp
public:
    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.

```cpp
template <class T>
struct FenwickTree2D {
    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 Segment Tree

The range should be of the form 2^p.

```cpp
template <class T, T(*op)(T, T), T ident>
struct SegmentTree {
    struct Node {
        T val;
        int l, r;
        Node(T _val, int _l, int _r) : val(_val), l(_l), r(_r) { };
    };
    int n;
    vector<Node> tree;
    SegmentTree(int p, vector<T> &init) : n(1 << p) { // Needs 2^p leafs
        tree.assign(2 * n, Node(ident, 0, n - 1));
        for (int j = 1; j < n; ++j) {
            int m = (tree[j].l + tree[j].r) / 2;
            tree[2*j].l = tree[j].l;
            tree[2*j].r = m;
            tree[2*j+1].l = m + 1;
            tree[2*j+1].r = tree[j].r;
        }
        for (int j = 2 * n - 1; j > 0; --j) {
            if (j >= n) tree[j].val = init[j - n];
            else tree[j].val = op(tree[2*j].val, tree[2*j+1].val);
        }
    }
    void update(int i, T val) {
        for (tree[i+n].val = val, i = (i+n)/2; i > 1; i /= 2)
            tree[i].val = op(tree[2*i].val, tree[2*i+1].val);
    }
    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++].val);
            if (!(r&1)) rhs = op(tree[r--].val, rhs);
        }
        return op(l == r ? op(lhs, tree[l].val) : lhs, rhs);
    }
};
```

2.6 Lazy Dynamic Segment Tree

```cpp
using T=int; using U=int;
T t_id; U u_id;
T merge(T a, T b){ return a+b; }
void join(U &a, U b){ a=a+b; }
struct Node {
    int l, r, lc, rc; T t; U u;
    Node(int l, int r, T t=t_id):l(l),r(r),lc(-1),rc(-1),t(t),u(u_id){}
};
T apply(const Node &n, int l=-1){ return merge(n.t,(l<0?n.r-n.l+1:l)*n.u); }
pair<T,T> split(T t, ll a, ll b){ return {t/(a+b)*a, t/(a+b)*b}; }
struct DynamicSegmentTree {
    vector<Node> tree;
    DynamicSegmentTree(int N) { tree.push_back({0,N-1}); }
    T query(int l, int r, int i = 0) {          // 0 <= l <= r < n
        auto &n = tree[i];
        if(l > n.r || r < n.l)   return t_id;   // <- disjunct, vv internal
        if(l <= n.l && n.r <= r) return apply(n);
        if(n.lc < 0) return apply(n, min(n.r,r) - max(n.l,l) + 1);
        n.t = apply(n);
        join(tree[n.lc].u, n.u);                // push the update
        join(tree[n.rc].u, n.u); n.u = u_id;    // and reset the update
        return merge(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(l > n.r || r < n.l) return;
        if(l <= n.l && n.r <= r){ join(n.u,u); return; }
        if(n.lc < 0 || n.rc < 0) {
            int m = (n.l + n.r) / 2;
            n.lc = tree.size(); n.rc = tree.size()+1;
            auto sp = split(n.t,m-n.l+1,n.r-m);
            tree.push_back({tree[i].l,  m, sp.first});
```

```
            tree.push_back({m+1,tree[i].r, sp.second});
        }       // DON'T use 'n' anymore, because tree may reallocate
        update(l,r,u,tree[i].lc); update(l,r,u,tree[i].rc);
        tree[i].t = merge(apply(tree[tree[i].lc]),apply(tree[tree[i].rc]));
    }
};
```

2.7 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.

```
struct Node {
    ll val, mx;
    int size, priority;
    bool rev = false;
    Node *l, *r;
    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;
}
```

2.8 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);
    }
};
```

2.9 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, *r;
    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;
}
```

2.10 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];
            }
    }
};
```

2.11 Suffix Array

Note: dont forget to invert the returned array. **Complexity:** $O(n \log^2 n)$

```
struct S { int l, r, p; };
bool operator<(const S &lhs, const S &rhs) {
    return lhs.l != rhs.l ? lhs.l < rhs.l : lhs.r < rhs.r;
}
bool operator==(const S &lhs, const S &rhs) {
    return lhs.l == rhs.l && lhs.r == rhs.r;
}

struct SuffixArray {
    string s;
    int n;
```

```
    vvi P;
    SuffixArray(string &_s) : s(_s), n(_s.length()) { construct(); }
    void construct() {
        vector<S> L(n, {0, 0, 0});
        P.push_back(vi(n, 0));
        for (int i = 0; i < n; ++i) P[0][i] = int(s[i]);
        for (int k = 1, cnt = 1; cnt / 2 < n; ++k, cnt *= 2) {
            P.push_back(vi(n, 0));
            for (int i = 0; i < n; ++i)
                L[i] = { P[k - 1][i], i + cnt < n
                    ? P[k - 1][i + cnt] : -1, i};
            sort(L.begin(), L.end());
            for (int i = 0; i < n; ++i)
                P[k][L[i].p] = (i > 0 && L[i] == L[i - 1]
                ? P[k][L[i - 1].p] : 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;
    }
};
```

2.12 Built-in datastructures

```
// Minimum Heap
#include <queue>
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;
using order_tree =
typedef 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] == 0) { // 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(){
        REP(u,V) 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(0),
        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() {
        REP(i,V) 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() {
        REP(u,V) 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, weight; };  // input edges
struct PQ{ int d, v; };         // distance and target
bool operator>(const PQ &l, const PQ &r){ return l.d > r.d; }
int dijkstra(vector<vector<Edge>> &edges, int s, int t) {
    vi dist(edges.size(),INF);
    priority_queue<PQ,vector<PQ>,greater<PQ>> pq;
    dist[s] = 0; pq.push({0, s});
    while (!pq.empty()) {
        auto d = pq.top().d, 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 Bellmann-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 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());
        }
    }
}
```

3.11 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;
    REP(u,edges.size()){
        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;
}
```

3.12 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*.

3.13 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;
    }
};
```

3.14 Heavy-Light decomposition

Complexity: $O(n)$

```
struct HLD {
    int V,T; vi &p; vvi &childs;    // Size; dfs-time; input parent/childs
    vi pr, size, heavy;     // path-root; size of subtrees; heavy child
    vi t_in, t_out;         // dfs in and out times
    HLD(vvi &childs, vi &p, int root = 0) :
        V(p.size()), T(0), p(p), childs(childs), pr(V,-1),
        size(V,-1), heavy(V,-1), t_in(V,-1), t_out(V,-1) {
            dfs(root); set_pr(root,0);
        }
    int dfs(int u){
        size[u] = 1; t_in[u] = T++;
        int m = -1, mi = -1, s;     // max, max index, size of subtree
        for(auto &v : childs[u]){
            size[u] += s = dfs(v);
            if(s > m) m=s, mi = v;
        }
        heavy[u] = mi; t_out[u] = T++; return size[u];
    }
    void set_pr(int u, int r){      // node, path root
        pr[u] = r;
        for(auto &v : childs[u]) set_pr(v, heavy[u] == v ? r : v);
    }
    bool is_parent(int p, int u){   // test whether p is a parent of u
        return t_in[p] <= t_in[u] && t_out[p] >= t_out[u];
    }
    int lca(int u, int v){
        while(!is_parent(pr[v],u)) v = p[pr[v]];
        while(!is_parent(pr[u],v)) u = p[pr[u]];
        return is_parent(u,v) ? u : v;
    }
};
```

4 Flow and Matching

4.1 Flow Graph

Structure used by the following flow algorithms.

```
struct S{
    int v;           // neighbour
    const int r;     // index of the reverse edge
    ll f;            // current flow
    const ll cap;    // capacity
    const ll cost;   // unit cost
    S(int v, int reverse_index, ll capacity, ll cost = 0) :
        v(v), r(reverse_index), f(0), cap(capacity), cost(cost) {}
};
struct FlowGraph : vector<vector<S>> {
    FlowGraph(size_t n) : vector<vector<S>>(n) {}
    void add_edge(int u, int v, ll capacity, ll cost = 0){
        operator[](u).emplace_back(v, operator[](v).size(), capacity, cost);
        operator[](v).emplace_back(u, operator[](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, ll 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(F(l),-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;
            REP(u,V) 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"
struct Q{ int u; ll c,w;}; // target, maxflow and total weight (cost)
bool operator>(const Q &l, const Q &r){return l.w > r.w;}
struct Edmonds_Karp_Dijkstra{
    FlowGraph &g; int V,s,t; vector<ll> pot;
    Edmonds_Karp_Dijkstra(FlowGraph &g, int s, int t) :
        g(g), V(g.size()), s(s), t(t), pot(V) {}
    pair<ll,ll> run() { // return pair<f, cost>
        ll maxflow = 0, cost = 0;
        fill(F(pot), LLINF); pot[s]=0; // Bellman-Ford for potentials
        REP(i,V-1) {
            bool relax = false;
            REP(u,V) if(pot[u] != LLINF) 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;
        }
        REP(u,V) if(pot[u] == LLINF) pot[u] = 0;
        while(true){
            priority_queue<Q,vector<Q>,greater<Q>> q;
            vector<vector<S>::iterator> p(V,g.front().end());
            vector<ll> dist(V, LLINF); ll f, tf = -1;
            q.push({s, LLINF, 0}); dist[s]=0;
            while(!q.empty()){
                auto u = q.top().u; ll 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++){
                    const auto &e = *it;
                    ll 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];
            }
            REP(u,V) if(dist[u]!=LLINF) 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:** $O(mw)$

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

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

        int w = mpref[m][mnext[m]++];
        if (wmatch[w] == -1) {
            mmatch[m] = w;
            wmatch[w] = m;
        } else {
            int mp = 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);
        }
    }
    return mmatch;
}
```

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

```
constexpr long double EPS = 1e-10;
using C = double;   // could be long long or long double
struct P {          // may also be used as a 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 dot(const P &p) const { return x * p.x + y * p.y; }
    C lensq() const { return x*x + y*y; }
    C len() const { return sqrt(lensq()); }
};

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> pts) {
    C sum = 0;
    REP(i,pts.size()) sum += det(pts[i], pts[(i+1)%pts.size()]);
    return sum;
}

double area(P p1, P p2, P p3) { return abs(det(p1, p2, p3)) / 2.0; }
double area(vector<P> polygon) { return abs(det(polygon)) / 2.0; }

// 1 when p1-p2-p3 is a left turn (when viewed from p1) [use EPS if needed]
int ccw(P p1, P p2, P p3) { C d = det(p1, p2, p3); return (d>0) - (d<0); }
struct S {
    P p1, p2;
    enum Type { Segment, Ray, Line } type;
    S(P p1 = 0, P p2 = 0, Type type = Line) : p1(p1), p2(p2), type(type) {}
    bool internal(P p) const {
        if(det(p1,p2,p) > EPS) return false; // not on a line
        switch(type){
        case Segment: return dist(p1, p) + dist(p, p2) - dist(p1,p2) <= EPS;
        case Ray: return dist(p,p2) - abs(dist(p1,p) - dist(p1,p2)) <= EPS;
        default: return true;
        }
    }
};
struct L{
    C a,b,c; // ax + by + c = 0
    L(C a = 0, C b = 0, C c = 0) : a(a), b(b), c(c) {}
    L(S s) : a(s.p2.y-s.p1.y), b(s.p1.x-s.p2.x),
    c(s.p2.x*s.p1.y - s.p2.y*s.p1.x) {}
    operator S(){
        S s; s.type = S::Line;
        if(abs(a)<EPS) s.p1 = {0, -c/b}, s.p2 = {1, -c/b};
        else s.p1 = {-c/a, 0}, s.p2 = {-(c+b)/a, 1};
        return s;
    }
};
struct Circle{ P p; C r; };
P project(S s, P p) {
    double l = (p-s.p1).dot(s.p2-s.p1)/double((s.p2-s.p1).dot(s.p2-s.p1));
    switch(s.type){
    case S::Segment: l = min(1.0, l);
    case S::Ray:     l = max(0.0, l);
    default:;
    }
    return s.p1 + (s.p2 - s.p1) * l;
}
pair<bool,P> intersect(const L &l1, const L &l2) {
    double x = l1.b*l2.c-l1.c*l2.b, y = l1.c*l2.a-l1.a*l2.c,
           z = l1.a*l2.b-l1.b*l2.a;
    return {z!=0, {x/z, y/z}};
}
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

```cpp
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]
        //  h[j], h[a]
    }
}
```

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

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:

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

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

7 Mathematics

7.1 Primes

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

```cpp
#include "numbertheory.cpp"
constexpr ll SIZE = 1e6+10;
bitset<SIZE + 1> bs;
vector<ll> primes;

void sieve() { // call at start in main!
    bs.set();
    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);
    }
}
```

```cpp
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 prime; ll exp;};
vector<Factor> factor(ll n) {
    vector<Factor> factors;
    for(const auto &prime : primes){
        if(n==1 || prime*prime > n) break;
        ll exp=0;
        while(n % prime == 0)
            n/=prime, exp++;
        if(exp>0)
            factors.push_back({prime,exp});
    }
    if (n != 1) factors.push_back({n,1});
    return factors;
}

vector<ll> mf(SIZE + 1, -1);        // mf[i]==i when prime
void sieve2() { // call at start in main!
    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);
    }
}

vector<Factor> factor2(ll n){
    vector<Factor> factors;
    while(n>1){
        if(factors.back().prime == mf[n]) factors.back().exp++;
        else factors.push_back({mf[n],1});
        n/=mf[n];
    }
    return factors;
}

ll numDiv(ll n) {
    ll divisors = 1;
    for(auto &&p : factor(n))
        divisors *= p.exp + 1;
    return divisors;
}

ll bin_pow(ll b, ll e){
    ll p = e==0 ? 1 : pow(b*b,e>>1);
    return p * p * (e&1 ? b : 1);
}

ll sumDiv(ll n) {
    ll sum = 1;
    for(const auto &p : factor(n))
        sum *= (pow(p.prime, p.exp+1) - 1) / (p.prime - 1);
    return sum;
}

ll EulerPhi(ll n) {
    ll ans = n;
    for(const auto &p : factor(n))
        ans -= ans / p.prime;
    return ans;
}

vector<ll> test_primes = {2,3,5,7,11,13,17,19,23}; // sufficient to 3.8e18
bool miller_rabin(const ll n){ // true when prime
    if(n<2) return false;
    if(n%2==0) return n==2;
    ll s = 0, d = n-1; // n-1 = 2^s * d
    while(d&1) s++, d/=2;
    for(auto a : test_primes){
        if(a > n-2) break;
        ll x = powmod(a,d,n);   // needs powmod with mulmod!
        if(x == 1 || x == n-1) continue;
        REP(i,s-1){
            x = mulmod(x,x,n);
            if(x==1) return false;
            if(x==n-1) goto next_it;
        }
        return false;
next_it:;
    }
    return true;
}
```

7.2 Number theoretic algorithms

```cpp
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));
}
```

```cpp

// (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;
}

// Finds a^n % m in O(lg n) time, ensure that a < m to avoid overflow!
ll powmod(ll a, ll n, ll m) {
    if (n == 0) return 1;
    if (n == 1) return a;
    ll aa = (a*a)%m; // use mulmod when b > 1e9
    if (n % 2 == 0) return powmod(aa, n / 2, m);
    return (a * powmod(aa, (n - 1) / 2, m)) % m;
}

// 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;
    }
}

// Chinese remainder theorem: finds z s.t. z % xi = ai. z is
// unique modulo M = lcm(xi). Returns (z, M), m = -1 on failure.
ii crm(ll x1, ll a1, ll x2, ll a2) {
    ll s, t, d;
    extended_euclid(x1, x2, s, t, d);
    if (a1 % d != a2 % d) return ii(0, -1);
    return ii(mod(s * a2 * x1 + t * a1 * x2, x1 * x2) / d, x1 * x2 / d);
}
ii crm(vi &x, vi &a){        // ii = pair<long,long>!
    ii ret = ii(a[0], x[0]);
    for (size_t i = 1; i < x.size(); ++i) {
        ret = crm(ret.second, ret.first, x[i], a[i]);
        if (ret.second == -1) break;
    }
    return ret;
}

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

7.3 Lucas' theorem

```cpp
#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;
}
```

7.4 Complex Numbers

Faster-than-built-in complex numbers

```cpp
struct Complex {
    long double u,v;
    Complex operator+(Complex r) const { return {u+r.u, v+r.v}; }
    Complex operator-(Complex r) const { return {u-r.u, v-r.v}; }
    Complex operator*(Complex r) const {
        return {u * r.u - v * r.v, u * r.v + v * r.u};
    }
    Complex operator/(Complex 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};
    }
    static
    Complex exp(complex<ld> c){ c = std::exp(c); return {c.real(), c.imag()
        }; }
};
```

7.5 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

```cpp
#define MY_PI 3.14159265358979323846
#include "../helpers/bitmasking.cpp"
#include <complex>
#include "complex.cpp"

// A.size() = N = 2^p
void fft(vector<Complex> &A, int N, int p, bool inv = false) {
    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) {
        Complex w, w_m = Complex::exp(complex<ld>(0, 2*MY_PI/m*(inv?-1:1)));
        for (int k = 0; k < N; k += m) {
            w = {1, 0};
            for (int j = 0; j < m / 2; ++j) {
                Complex 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) for (int i = 0; i < N; ++i) {
        A[i].u /= N; A[i].v /= N;
    }
}

void convolution(vector<Complex> &A,vector<Complex> &B,vector<Complex> &C){
    // Pad with zeroes
    int N = 2 * max(next_power_of_2(A.size()), next_power_of_2(B.size()));
    A.reserve(N); B.reserve(N); C.reserve(N);
    for (int i = A.size(); i < N; ++i) A.push_back({0, 0});
    for (int i = B.size(); i < N; ++i) B.push_back({0, 0});
    int p = int(log2(N) + 0.5);
    // Transform A and B
    fft(A, N, p, false);
    fft(B, N, p, false);
    // Calculate the convolution in C
    for (int i = 0; i < N; ++i) C.push_back(A[i] * B[i]);
    fft(C, N, p, true);
}

void square_inplace(vector<Complex> &A) {
    int N = 2 * next_power_of_2(A.size());
    A.reserve(N);
    for (int i = A.size(); i < N; ++i) A.push_back({0, 0});
    int p = int(log2(N) + 0.5);
    fft(A, N, p, false);
    for (int i = 0; i < N; ++i) A[i] = A[i] * A[i];
    fft(A, N, p, true);
}
```

7.6 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;
}
```

7.7 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;
    }
    static M raise(const M &m, int n) {
        if(n == 0) return id();
        if(n == 1) return m;
        auto r = (m*m).raise(n / 2);
        return (n%2 ? m*r : r);
    }
};
```

7.8 Simplex algorithm

Maximize $c^t x$ subject to $Ax \le b$ and $x \ge 0$. $A[m \times n]$, $b[m]$, $c[n]$, $x[n]$. 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];
    }
};
```

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;
        vector<int> match;
        Node() {
            for (int i = 0; i < ALPHABET_SIZE; ++i) child[i] = -1;
        }
    };
    vector <Node> a;
    AC_FSM() { a.push_back(Node()); }
    void construct_automaton(vector<string> &words) {
        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; k < ALPHABET_SIZE; ++k) {
                if (a[r].child[k] != -1) {
                    q.push(a[r].child[k]);
                    int v = a[r].failure;
                    while (a[v].child[k] == -1) v = a[v].failure;
                    a[a[r].child[k]].failure = a[v].child[k];
                    for (int w : a[a[v].child[k]].match)
                        a[a[r].child[k]].match.push_back(w);
                }
            }
        }
    }

    void aho_corasick(string &sentence, vector<string> &words,vvi &matches){
        matches.assign(words.size(), vector<int>());
        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 (int w : a[state].match)
                matches[w].push_back(i - words[w].length() + 1);
        }
    }
};
```

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 Miscellaneous

9.1 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] =  lis[l - 2];
        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;
}
```

9.2 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 Helpers

10.1 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
}
```

10.2 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;
}
```

10.3 Bitmasking

```
#ifdef _MSC_VER
#define popcount(x) __popcnt(x)
#else
#define popcount(x) __builtin_popcount(x)
#endif

bool bit_set(int mask, int pos) {
    return ((mask & (1 << pos)) != 0);
}

// Iterate over all subsets of a set of size N
for (int mask = 0; mask < (1 << N); ++mask) {
    // Decode mask here
}

// Iterate over all k-subsets of a set of size N
```

```
int mask = (1 << k) - 1;
while (!(mask & 1 << N)) {
    // Decode mask here
    int lo = mask & ~(mask - 1);
    int lz = (mask + lo) & ~mask;
    mask |= lz;
    mask &= ~(lz - 1);
    mask |= (lz / lo / 2) - 1;
}

// Iterate over all subsets of a subset
int mask = givenMask;
do {
    // Decode mask here
    mask = (mask - 1) & givenMask;
} while (mask != givenMask);

// The two functions below are used in the FFT:
inline int next_power_of_2(int x) {
    x = (x - 1) | ((x - 1) >> 1);
    x |= x >> 2; x |= x >> 4;
    x |= x >> 8; x |= x >> 16;
    return x + 1;
}

inline int brinc(int x, int k) {
    int 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;
}
```

10.4 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 Strategies

11.1 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

11.2 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

11.3 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)

11.4 RTE

- Typos
- Off by one error (in array index of loop bound)
- return 0 at end of program