Team Code Reference

Contents

1 Template

1.1 C++ Template

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <bitset>
#include <algorithm>
#include <functional>
#include <string>
#include <string.h>  // Include for memset!
#include <complex>
#include <random>
#define _USE_MATH_DEFINES
#include <math.h>

#define INF 2000000000              // 9
#define LLINF 9000000000000000000LL // 18
#define LDINF 1e200

using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<ii> vii;
typedef vector<vii> vvii;
typedef vector<bool> vb;
typedef long long ll;
typedef long double ld;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    // Solve

    return 0;
}
```

1.2 Java Template*

2 Data Structures

2.1 Union Find

```cpp
vii pset;
int psets;

```

```
class UnionFind {
private:
    vi parent, rank, setSize;
    int setCount;
public:
    UnionFind(int N) {
        setSize.assign(N, 1);
        setCount = N;
        rank.assign(N, 0);
        parent.assign(N, 0);

        for (int i = 0; i < N; ++i) parent[i] = i;
    }

    int find_set(int i) {
        return (parent[i] == i) ? i : (parent[i] = find_set(parent[i]));
    }

    bool are_same_set(int i, int j) {
        return (find_set(i) == find_set(j));
    }

    void union_set(int i, int j) {
        if ((i = find_set(i)) == (j = find_set(j))) return;
        setCount--;
        if (rank[i] > rank[j]) {
            parent[j] = i;
            setSize[i] += setSize[j];
        } else {
            parent[i] = j;
            setSize[i] += setSize[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;
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.

```
template <class T>
class FenwickTree2D {
private:
    vector< vector<T> > tree;
    int n;
public:
    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*

2.6 Range Minimum Query*

2.7 Segment tree*

2.8 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];
            }
    }
};
```

2.9 Suffix array*

2.10 Heavy-Light decomposition*

2.11 Pareto Front*

3 Basic Graph algorithms

3.1 Edge Classification

Complexity: $O(V + E)$

```
vvi edges;
vi color, parent;

void classify(int u) {
    color[u] = 1;
    for (int v : edges[u]) {
        if (color[v] == 0) {
            // u -> v is a tree edge
            parent[v] = u;
            classify(v);
        } else if (color[v] == 1) {
            if (v == parent[u]) {
                // u -> v, v -> u 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;
}
```

3.2 Articulation points and bridges*

3.3 Topological sort

Complexity: $O(V + E)$

```
vi sorted;
vb visited;
int s_ix = 0;
vvi edges;

void visit(int u) {
    visited[u] = true;
    for (int v : edges[u])
```

```
        if (!visited[v]) visit(v);
    sorted[s_ix--] = u;
}

void topo_sort() {
    s_ix = edges.size() - 1;
    sorted = vi(edges.size());
    visited = vb(edges.size(), false);
    for (int i = 0; i < edges.size(); ++i) {
        if (!visited[i]) visit(i);
    }
}
```

3.4 Kruskal's algorithm

Complexity: $O(E \log_2 V)$
Dependencies: Union Find

```
// Edges are given as (weight, (u, v)) triples.
int kruskal(vector< pair<int, ii> > &edges, int V) {
    sort(edges.begin(), edges.end());
    int cost = 0, count = 0;
    UnionFind uf(V);
    for (pair<int, ii> e : edges) {
        if (!uf.are_same_set(e.second.first, e.second.second)) {
            // (w, (u, v)) is part of the MST
            cost += e.first;
            uf.union_set(e.second.first, e.second.second);
            if ((++count) == V - 1) break;
        }
    }
    return cost;
}
```

3.5 Prim's algorithm

Complexity: $O(E \log_2 V)$

```
typedef pair<int, ii> iii;
// Adjacency list given as (endpoint, weight)
ll prim(vvii& adj, vii& tree) {
    ll tc = 0; vb intree(adj.size(), false);
    priority_queue<iii, vector<iii>, greater<iii> > pq;

    intree[0] = true;
    for (ii e : adj[0]) pq.push(iii(e.second, ii(0, e.first)));

    while (!pq.empty()) {
        int c = pq.top().first; ii e = pq.top().second; pq.pop();
        if (intree[e.second]) continue;
        intree[e.second] = true;
        tc += c; tree.push_back(e);
        for (ii e2 : adj[e.second]) {
            if (!intree[e2.first])
                pq.push(iii(e2.second, ii(e.second, e2.first)));
        }
    }
    return tc;
}
```

3.6 Biconnected components*

3.7 Strongly connected components*

3.8 Kosaraju's algorithm*

3.9 Dijkstra's algorithm

Complexity: $O((V + E) \log_2 V)$

```
// Input is an edge list with a vector for each vertex,
// containing a list of (endpoint, weight) edges (ii's).
void dijkstra(vvii edges, int source) {
    vi dist(edges.size(), INF);
    priority_queue<ii, vector<ii>, greater<ii>> pq;
    dist[source] = 0; pq.push(ii(0, source));

    while (!pq.empty()) {
        ii top = pq.top(); pq.pop();
        int u = top.second, d = top.first;
        // <= Goal check on u here.
        if (d == dist[u]) {
            for (ii it : edges[u]) {
                int v = it.first, d_uv = it.second;
                if (dist[u] + d_uv < dist[v]) {
                    dist[v] = dist[u] + d_uv;
                    pq.push(ii(dist[v], v));
}   }   }   }   }
```

3.10 Bellmann-Ford algorithm

Returns `true` if the graph has no negative cycles.
Complexity: $O(VE)$

```
// Edge list as in with Dijkstra's (see above)
bool bellmann_ford(vvii edges, int source, vi &dist) {
    dist.assign(edges.size(), INF); dist[source] = 0;
    for (int iter = 0; iter < edges.size() - 1; ++iter)
        for (int u = 0; u < edges.size(); ++u)
            for (ii e : edges[u])
                dist[e.first] = min(dist[e.first], dist[u] + e.second);
    for (int u = 0; u < edges.size(); ++u)
        for (ii e : edges[u])
            if (dist[e.first] > dist[u] + e.second)
                return false;
    return true;
}
```

3.11 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.
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)
                adj[a][c] = min(adj[a][c], adj[a][b] + adj[b][c]);
}
```

3.12 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());
        }
    }
}
```

3.13 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*.

4 Flow Algorithms

4.1 Flow Network

Generic flow network used by the algorithms in this section. Should not require any modifications. *Note:* Get the reverse of `e[i]` as `e[i ^1]`. Don't forget to `cleanup()` afterwards.

```
struct FlowNetwork {
    struct edge {
        int v, nxt; ll cap, flo;
        edge(int _v, ll _cap, int _nxt) : v(_v), cap(_cap), nxt(_nxt), flo
            (0) { }
    };
    int n, edge_count = 0, *h;
    vector<edge> e;
    FlowNetwork(int V, int E = 0) : n(V) {
        e.reserve(2 * (E == 0 ? V : E));
```

```
        memset(h = new int[V], -1, n * sizeof(int));
    }
    void add_edge(int u, int v, ll uv_cap, ll vu_cap = 0) {
        e.push_back(edge(v, uv_cap, h[u])); h[u] = edge_count++;
        e.push_back(edge(u, vu_cap, h[v])); h[v] = edge_count++;
    }
    void cleanup() { delete[] h; }
    // Only copy what is needed:
    ll edmonds_karp(int s, int t);
    ll dinic(int s, int t);
    ll dinic_augment(int s, int t, int* d, ll cap);
    ll push_relabel(int s, int t);
    ll infer_mincut(int s);
    void infer_mincut_dfs(int u, vb& vs);
};
```

4.2 Edmonds-Karp algorithm

Complexity: $O(VE^2)$
Dependencies: Flow Network

```
#define MAXV 2000
ll FlowNetwork::edmonds_karp(int s, int t) {
    int v, p[MAXV], q[MAXV]; ll f = 0, c[MAXV];
    while (true) {
        memset(p, -1, n * sizeof(int));
        int i, u = -1, l = 0, r = 0;
        c[s] = LLINF; p[q[r++] = s] = -2; // -2 == source, -1 == unvisited
        while (l != r && u != t) {
            for (u = q[l++], i = h[u]; i != -1; i = e[i].nxt) {
                if (e[i].cap > e[i].flo && p[v = e[i].v] == -1) {
                    p[q[r++] = v] = i;
                    c[v] = min(c[u], e[i].cap - e[i].flo);
        }   }   }
        if (p[t] == -1) break;
        for (i = p[t]; i != -2; i = p[e[i ^ 1].v]) {
            e[i].flo += c[t]; e[i ^ 1].flo -= c[t];
        }
        f += c[t];
    }
    return f;
}
```

4.3 Dinic's algorithm

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

```
#define MAXV 5000
ll FlowNetwork::dinic_augment(int s, int t, int* d, ll cap) {
    if (s == t) return cap;
    ll f = 0, df = 0;
    for (int i = h[s]; i != -1; i = e[i].nxt) {
        if (e[i].cap > e[i].flo && d[s] + 1 == d[e[i].v]) {
            f += (df = dinic_augment(e[i].v, t, d, min(cap, e[i].cap - e[i].
                flo)));
            e[i].flo += df;
            e[i ^ 1].flo -= df;
            if((cap -= df) == 0) break;
    }   }
    return f;
}

ll FlowNetwork::dinic(int s, int t) {
    int q[MAXV], d[MAXV]; ll f = 0;
    while (true) {
        memset(d, -1, n * sizeof(int));
        int l = 0, r = 0, u = -1, i;
        d[q[r++] = s] = 0;
        while (l != r && u != t)
            for (u = q[l++], i = h[u]; i != -1; i = e[i].nxt)
                if (e[i].cap > e[i].flo && d[e[i].v] == -1)
                    d[q[r++] = e[i].v] = d[u] + 1;
        if (d[t] == -1) break;
        f += dinic_augment(s, t, d, LLINF);
    }
    return f;
}
```

4.4 Push-relabel algorithm*

4.5 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 FlowNetwork::infer_mincut_dfs(int u, vb& vs) {
    vs[u] = true;
    for (int i = h[u]; i != -1; i = e[i].nxt) {
        if (e[i].cap > e[i].flo && !vs[e[i].v])
            infer_mincut_dfs(e[i].v, vs);
}   }

ll FlowNetwork::infer_mincut(int s) {
    vb vs(n, false);
    infer_mincut_dfs(s, vs);
    ll c = 0;
    for (int i = 0; i < e.size(); ++i) {
        if (vs[e[i ^ 1].v] && !vs[e[i].v]) {
            // The edge e[i ^ 1].v -> e[i].v,
            // given as e[i], is in the min cut.
            c += e[i].cap;
    }   }
    return c;
}
```

5 Combinatorics & Probability

5.1 Essentials*

5.2 Hopcroft-Karp algorithm*

5.3 Hungarian algorithm*

5.4 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;
}
```

5.5 Meet in the Middle

Sufficient for $2 \leq n \leq 14$.
Complexity: $O(n^2\binom{n}{n/2}\left(\frac{n}{2}\right)!)$

```
#define MAX_N 15
ll d[MAX_N][MAX_N];

ll meet_in_the_middle(int n) {
    int half = (n - 2) / 2, otherhalf = n - 2 - half;
    vi leftroute(half, 0), rightroute(otherhalf, 0);
    ll shortest = LLINF;

    for (int m = 1; m < n; ++m) {
        int mask = (1 << half) - 1;
        while (!(mask & 1 << (n - 2))) {
            int l = 0, r = 0, p = 0;
            for (int v = 1; v < n; ++v) {
                if (v == m) continue;
                if (bit_set(mask, p++)) leftroute[l++] = v;
                else                    rightroute[r++] = v; }

            ll lmin = LLINF, rmin = LLINF;
            do{ ll routelength = d[0][leftroute.empty() ? m : leftroute[0]];
                for (int i = 1; i < half; ++i)
                    routelength += d[leftroute[i - 1]][leftroute[i]];
                if (!leftroute.empty())
                    routelength += d[leftroute[half - 1]][m];
                lmin = min(lmin, routelength);
            } while (next_permutation(leftroute.begin(), leftroute.end()));

            do{ ll routelength = d[m][rightroute.empty() ? 0 : rightroute
                [0]];
                for (int i = 1; i < otherhalf; ++i)
                    routelength += d[rightroute[i - 1]][rightroute[i]];
                if (!rightroute.empty())
                    routelength += d[rightroute[otherhalf - 1]][0];
                rmin = min(rmin, routelength);
            } while (next_permutation(rightroute.begin(), rightroute.end()))
                ;

            shortest = min(shortest, lmin + rmin);

            if ((mask != 0)) {
                int lo = mask & ~(mask - 1);
                int lz = (mask + lo) & ~mask;
                mask |= lz;
                mask &= ~(lz - 1);
                mask |= (lz / lo / 2) - 1;
            } else break;
    }   }
    return shortest;
}
```

5.6 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: $O(d^2 log_2(d) log_2(log_2(d)))$

```
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 (ll)ceil(a / (ld)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)};
  }
}
```

5.7 2-SAT*

6 Computational geometry

6.1 Essentials

```
#define EPSILON 1e-6

// Coordinate type, change to long long or double when necessary.
typedef int coord;

struct point {
public:
    coord x, y;
    point() {}
    point(coord x, coord y) : x(x), y(y)   {}
    point(const point &p) : x(p.x), y(p.y) {}
    point operator+ (const point &p) const { return point(x + p.x, y + p.y); }
    point operator- (const point &p) const { return point(x - p.x, y - p.y); }
    point operator* (double c) const { return point((coord)(x * c), (coord)(y * c)); }
    point operator/ (double c) const { return point((coord)(x / c), (coord)(y / c)); }
    bool  operator< (const point &r) const { return (y != r.y ?
                                                      (y < r.y) : (x > r.x)); }
    bool  operator==(const point &r) const { return (y == r.y  && x == r.x); }
};
struct line {
    point p1, p2;
    line() {}
    line(point p1, point p2) : p1(p1), p2(p2) {}
    line(const line &l) : p1(l.p1), p2(l.p2) {}
};
enum LineType { LINE, RAY, SEGMENT };

coord dot(point p1, point p2) { return p1.x * p2.x + p1.y * p2.y; }
coord lensq(point p1, point p2) {
    return (p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y);
}

coord det(coord x1, coord y1, coord x2, coord y2) { return x1 * y2 - x2 * y1; }
coord det(point p1, point p2) { return p1.x * p2.y - p1.y * p2.x; }
coord det(point p1, point p2, point origin) {
    return (p1.x - origin.x)*(p2.y - origin.y) - (p1.y - origin.y)*(p2.x - origin.x);
}
coord det(vector<point> pts) {
    coord sum = 0;
    for(int i = 0; i < pts.size(); ++i)
        sum += det(pts[i], pts[(i + 1) % pts.size()]);
    return sum;
}

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

int seq(point p1, point p2, point p3) {
    coord d = det(p1, p2, p3);
    return (d < 0 ? -1:     // Right turn
            d > 0 ?  1:     // Left  turn
                     0);    // Points are colinear
}

point project(line l, point p, LineType type) {
    double lambda = dot(p - l.p1, l.p2 - l.p1)/((double)dot(l.p2 - l.p1, l.p2 - l.p1));
    switch(type){
        case LineType.SEGMENT: lambda = min(1.0, lambda);
        case LineType.RAY:     lambda = max(0.0, lambda);
        default: break;
    }
    return l.p1 + (l.p2 - l.p1) * lambda;
}

bool intersect_lines(line l1, line l2, double* lambda, LineType type) {
    // Intersection point can be reconstructed as l1.p1 + lambda * (l1.p2 - l1.p1).
    // Returns false if the lines are parallel, handle coincidence in advance.
    coord s1x, s1y, s2x, s2y;
    s1x = l1.p2.x - l1.p1.x;    s1y = l1.p2.y - l1.p1.y;
    s2x = l2.p2.x - l2.p1.x;    s2y = l2.p2.y - l2.p1.y;
    coord denom = det(s1x, s1y, s2x, s2y);
```

```
    if (denom == 0) return false;
    else{
        double l = det(s1x, s1y, l1.p1.x - l2.p1.x, l1.p1.y - l2.p1.y)/((
            double)denom),
               m = det(s2x, s2y, l1.p1.x - l2.p1.x, l1.p1.y - l2.p1.y)/((
                   double)denom);
        switch(type){
            case LineType.SEGMENT:  if(l > 1 || m > 1) return false;
            case LineType.RAY:      if(l < 0 || m < 0) return false;
            default: break;
        }
        *lambda = l;
        return true;
    }
}
```

6.2 Convex Hull

Complexity: $O(n \log_2 n)$
Dependencies: Geometry Essentials

```
point pivot;

bool angle_compare(point p1, point p2) {
    if (det(pivot, a, b) == 0) return lensq(pivot, a) < lensq(pivot, b);
    int d1x = a.x - pivot.x, d1y = a.y - pivot.y,
        d2x = b.x - pivot.x, d2y = b.y - pivot.y;
    return (atan2((double)d1y, (double)d1x) - atan2((double)d2y, (double)d2x
        )) < 0;
}

vector<point> graham_scan(vector<point> pts) {
    int i, P0 = 0, N = pts.size();
    for (i = 1; i < N; ++i) {
        if (pts[i] < pts[P0].y) P0 = i;
    }
    pivot = pts[P0];
    pts[P0] = pts[0];
    pts[0] = pivot;
    sort(++pts.begin(), pts.end(), angle_compare);

    stack<point> S;
    point prev, now;
    S.push(pts[N - 1]);
    S.push(pts[0]);
    i = 1;
    while (i < N) { // Requires 3+ points to work
        now = S.top(); S.pop();
        prev = S.top(); S.push(now);
        if (seq(prev, now, pts[i]) > 0) { // Change to >= to allow colinear
            points
            S.push(pts[i]);
            i++;
        } else S.pop();
    }

    vector<point> ch_pts;
    while(!S.empty()) ch_pts.push_back(S.top()); S.pop();
    ch_pts.pop_back();
    return ch_pts;
}
```

6.3 Halfspace intersections*

7 Mathematics

7.1 Primes

```
ll _sieve_size;
bitset<10000010> bs;
vi primes;

void sieve(ll upperbound) {
    _sieve_size = upperbound + 1;
    bs.reset(); bs.flip();
    bs.set(0, false); bs.set(1, false);
    for (ll i = 2; i <= _sieve_size; ++i) {
        for (ll j = i * i; j <= _sieve_size; j += i) bs.set((size_t)j, false
            );
        primes.push_back((int)i);
    }
}

bool is_prime(ll N) {    // Only works for N <= primes.last^2
    if (N < _sieve_size) return bs.test(N);
    for (int i = 0; i < primes.size(); ++i) if (N % primes[i] == 0) return
        false;
    return true;
}

vi prime_factors(int N) {
    int PFD_idx = 0, PF = primes[PF_idx]; vi factors;
    while (N != 1 && PF * PF <= N) {
        while (N % PF == 0) { N /= PF; factors.push_back(PF); }
        PF = primes[++PF_idx];
    }
    if (N != 1) factors.push_back(N);
    return factors;
}

ll totient(ll N) {
    vi factors = prime_factors(N);
    vi::iterator new_end = unique(factors.begin(), factors.end());
    ll result = N;
    for (vi::iterator i = factors.begin(); i != new_end; ++i)
        result = result - result / (*i);
    return result;
}
```

7.2 Number theoretic algorithms

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

// Finds x, y s.t. ax + by = d = gcd(a, b).
void extended_euclid(int a, int b, int &x, int &y, int &d) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a / b;
        int 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
int mod_inverse(int a, int n) {
    int x, y, d;
    extended_euclid(a, n, x, y, d);
    return (d > 1 ? -1 : mod(x, n);
}

// Solve ax + by = c, returns false on failure.
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
    int 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(int x1, int a1, int x2, int a2) {
    int s, t, d;
    extended_euclid(x, y, s, t, d);
    if (a % d != b % 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 ret = ii(a[0], x[0]);
    for (int i = 1; i < x.size(); ++i) {
        ret = crm(ret.second, ret.first, x[i], a[i]);
        if (ret.second == -1) break;
    }
    return ret;
}
```

7.3 Complex Numbers

Faster-than-built-in complex numbers

```
typedef pair<ld, ld> cmpx;
cmpx cadd(cmpx lhs, cmpx rhs) {
    return cmpx(lhs.first + rhs.first, lhs.second + rhs.second);
}
cmpx csub(cmpx lhs, cmpx rhs) {
    return cmpx(lhs.first - rhs.first, lhs.second - rhs.second);
}
cmpx cmul(cmpx lhs, cmpx rhs) {
    return cmpx(lhs.first * rhs.first - lhs.second * rhs.second,
                lhs.first * rhs.second + lhs.second * lhs.first);
}
cmpx cdiv(cmpx lhs, cmpx rhs) {
    ld a = lhs.first, b = lhs.second,
       c = rhs.first, d = rhs.second;
    return cmpx((a * c + b * d) / (c * c + d * d),
                (b * c - a * d) / (c * c + d * d));
}
cmpx cexp(complex<ld> e) {
    e = exp(e);
    return cmpx(real(e), imag(e));
}
```

7.4 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: $O(n \log_2 n)$
Dependencies: Bitmasking, Complex Numbers

```
#define MY_PI 3.14159265358979323846

// A.size() = N = 2^p
void fft(vector<cmpx>& 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) {
        cmpx w_m = cexp(complex<ld>(0, 2 * MY_PI / m * (inv ? -1 : 1))), w;
        for (int k = 0; k < N; k += m) {
            w = cmpx(1, 0);
            for (int j = 0; j < m / 2; ++j) {
                cmpx t = cmul(w, A[k + j + m / 2]);
                A[k + j + m / 2] = csub(A[k + j], t);
                A[k + j] = cadd(A[k + j], t);
                w = cmul(w, w_m);
            }
        }
    }
    if (inv) for (int i = 0; i < N; ++i) {
        A[i].first /= N; A[i].second /= N;
    }
}

void convolution(vector<cmpx>& A, vector<cmpx>& B, vector<cmpx>& 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);
    for (int i = B.size(); i < N; ++i) B.push_back(0);
    int p = (int)round(log2(N));
    // 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(cmul(A[i], B[i]));
    fft(C, N, p, true);
}
```

7.5 BigInteger*

7.6 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 Matrix {
    T m[N][N];
    Matrix() { ITERATE_MATRIX(N) m[c][r] = 0; }
    Matrix(Matrix& o) { ITERATE_MATRIX(N) m[c][r] = o.m[c][r]; }
    static Matrix<T, N> identity() {
        Matrix<T, N> I;
        for (int i = 0; i < N; ++i) I.m[i][i] = 1;
        return I;
    }
    static Matrix<T, N> multiply(Matrix<T, N> lhs, Matrix<T, N> rhs) {
        Matrix<T, N> out;
        ITERATE_MATRIX(N)
            for (int i = 0; i < N; ++i)
                out.m[c][r] += lhs.m[i][r] * rhs.m[c][i];
        return out;
    }
    Matrix<T, N> raise(int n) {
        if (n == 0) return Matrix<T, N>::identity();
        if (n == 1) return Matrix<T, N>(*this);
        if (n == 2) return Matrix<T, N>::multiply(*this, *this);
        if (n % 2 == 0)
            return Matrix<T, N>::multiply(*this, *this).raise(n / 2);
        return Matrix<T, N>::multiply(*this,
            Matrix<T, N>::multiply(*this, *this).raise((n - 1) / 2));
    }
};
```

8 Strings

8.1 Knuth Morris Pratt

Complexity: $O(n+m)$

```
void compute_prefix_function(string& word, vi& pi) {
    pi = vector<int>(word.length());
    pi[0] = -1; pi[1] = 0;
    int i = 2, k = 0;

    while (i < pi.size()) {
        if (word[i - 1] == word[k]) {
            pi[i] = k + 1;
            i++; k++;
        }
        else if (k > 0) k = pi[k];
        else { pi[i] = 0; i++; }
    }
}

void knuth_morris_pratt(string& sentence, string& word) {
    int q = -1; vi pi;
    compute_prefix_function(word, pi);
    for (int i = 0; i < sentence.length(); ++i) {
        while (q >= 0 && word[q + 1] != sentence[i]) q = pi[q];
        if (word[q + 1] == sentence[i]) q++;
        if (q == word.length() - 1) {
            // Match at position (i - word.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: $O(n+m+k)$

```cpp
template <int ALPHABET_SIZE, int (*mp)(char)>
class 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;
public:
    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, vector<vector<int> >& 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);
        }
    }
};
```

9 Helpers

9.1 Golden Section Search

For a discrete search: use binary search on the difference of successive elements, see the section on Binary Search.

Complexity: $O(\log_2 1/\epsilon)$

```cpp
#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
}
```

9.2 Binary Search

Complexity: $O(\log_2 n), O(\log_2 1/\epsilon)$

```cpp
# 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;
}
```

9.3 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;
}
```

9.4 QuickSelect

Running time is expected, quadratic in the worst case. Alternatingly breaks ties left and right, so it should be pretty resilient to edge cases. Note that the vector is changed in the process. Recursion depth is $O(\log_2 n)$.
Complexity: $O(n)$

```
template<class T>
T quickselect(vector<T>& v, int l, int r, int k) {
    int p = l + (rand() % (r - l));
    swap(v[l], v[p]);
    bool alt = false; p = l + 1;
    for (int j = l + 1; j < r; ++j) {
        if (alt = !alt) {
                if (v[j] <  v[l]) swap(v[p++], v[j]);
        } else if (v[j] <= v[l]) swap(v[p++], v[j]);
    }
    swap(v[l], v[--p]);

    if (p == k) return v[k];
    if (p >  k) return quickselect(v, l, p, k);
    if (p <  k) return quickselect(v, p + 1, r, k);
}
```