Competitive Programming Algorithms

Contents

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g., Enumerating permutations (Section 3.2)
$\leq [17..19]$	$O(2^n \times n^2)$	e.g., DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g., DP with bitmask technique (Book 2)
$\leq [24..26]$	$O(2^n)$	e.g., try 2^n possibilities with $O(1)$ check each
≤ 100	$O(n^4)$	e.g., DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 450	$O(n^3)$	e.g., Floyd-Warshall (Section 4.5)
$\leq 1.5K$	$O(n^{2.5})$	e.g., Hopcroft-Karp (Book 2)
$\leq 2.5K$	$O(n^2 \log n)$	e.g., 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g., Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 200K$	$O(n^{1.5})$	e.g., Square Root Decomposition (Book 2)
$\leq 4.5M$	$O(n \log n)$	e.g., Merge Sort (Section 2.2)
$\leq 10M$	$O(n \log \log n)$	e.g., Sieve of Eratosthenes (Book 2)
$\leq 100M$	$O(n), O(\log n), O(1)$	Most contest problem have $n \leq 1M$ (I/O bottleneck)

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800

n	11	12	13	14	15	16	17
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14

n	20	25	30	40	50	100	150	171
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX

1 Introduction and Miscellaneous

1.1 Template

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

int main() {
    cin.tie(0)->sync_with_stdio(0);

    int n;
    cin >> n;
}
```

1.2 .bashrc

```bash
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
    -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>
```

1.3 Generate Files

```bash
for i in {B..L}.cpp; do cp "A.cpp" "$i"; done
```

1.4 Binary Search

```cpp
int binarysearch(function<bool(int)> f) {
    int lo = 0, hi = 100000, bestSoFar = -1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (f(mid)) {
            bestSoFar = mid;
            hi = mid - 1;
        } else lo = mid + 1;
    }
    return bestSoFar;
}
```

1.5 Base 10 to Base m

```cpp
char a[16] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};

string tenToM(int n, int m) {
    int temp = n;
    string result = "";
    while (temp != 0) {
        result = a[temp % m] + result;
        temp /= m;
    }

    return result;
}
```

1.6 Coordinate Compression

```
// coordinates -> (compressed coordinates).
map<int, int> coordMap;

void compress(vector<int>& values) {
    for (int v : values) coordMap[v] = 0;
    int cId = 0;
    for (auto it = coordMap.begin(); it != coordMap.end(); ++it) it->second = cId++;
    for (int& v : values) v = coordMap[v];
}
```

2 Data Structures

2.1 Order Statistic Tree (Set)

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

template<class T>
using OST = tree<T, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
    ordered_set;

int main() {
    OST<int> t, t2;
    t.insert(8);
    auto it = t.insert(10).first;
    assert(it == t.lower_bound(9));
    assert(t.order_of_key(10) == 1);
    assert(t.order_of_key(11) == 2);
    assert(*t.find_by_order(0) == 8);
    t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

2.2 Order Statistic Tree (Map)

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

template<class T, class E>
using OST = tree<T, E, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;
```

2.3 Union Find

```
struct UF {
    vector<int> e;
    UF(int n) : e(n, -1) {}
    bool sameSet(int a, int b) { return find(a) == find(b); }
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : e[x] = find(e[x]); }
    bool join(int a, int b) {
        a = find(a), b = find(b);
```

```
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        e[a] += e[b];
        e[b] = a;
        return true;
    }
};
```

2.4 Sparse Table

```
const int N = 100000;
const int LOGN = 18;

int a[N];
// sparseTable[l][i] = max a[i..i+2^l)
int sparseTable[LOGN][N];

void precomp(int n) {
    // level 0 is the array itself
    for (int i = 0; i < n; i++) sparseTable[0][i] = a[i];

    for (int l = 1; l < LOGN; l++) {  // inner loop does nothing if 2^l > n
        int w = 1 << (l - 1);          // 2^(l-1)

        // a[i,i+2w) is made up of a[i,i+w) and a[i+w,i+2w)
        for (int i = 0; i + 2 * w <= n; i++)
            sparseTable[l][i] = max(sparseTable[l - 1][i], sparseTable[l - 1][i + w]);
    }
}
```

2.5 Segment Tree

```
struct Tree {
    typedef int T;
    static constexpr T unit = INT_MIN;
    T f(T a, T b) { return max(a, b); }  // any associative function

    vector<T> s;
    int n;

    Tree(int n = 0, T def = unit) : s(2 * n, def), n(n) {}

    void update(int pos, T val) {
        for (s[pos += n] = val; pos /= 2;) s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
    }

    T query(int b, int e) {  // query [b, e)
        T ra = unit, rb = unit;
        for (b += n, e += n; b < e; b /= 2, e /= 2) {
            if (b % 2) ra = f(ra, s[b++]);
            if (e % 2) rb = f(s[--e], rb);
        }
        return f(ra, rb);
    }
};
```

2.6 Segment Tree on Trees

```
vector<int> children[N];

int indexInRangeTree[N], startRange[N], endRange[N];
int totId;

void compute_tree_ranges(int v) {
    indexInRangeTree[v] = startRange[v] = totId++;
    for (int w : children[v]) compute_tree_ranges(w);
    endRange[v] = totId;
}

void update_node(Tree &t, int id, int v) { t.update(indexInRangeTree[id], v); }
long long query_subtree(Tree &t, int id) { return t.query(startRange[id], endRange[id]); }
```

3 Dynamic Programming

3.1 Knapsack

```
int dp[N + 2][S + 1];

for (int i = N; i >= 1; --i) {
    for (int r = 0; r <= S; ++r) {
        int m = dp[i + 1][r];
        if (r - s[i] >= 0) m = max(m, dp[i + 1][r - s[i]] + v[i]);
        dp[i][r] = m;
    }
}
```

3.2 Bitsets

```
// for all sets
for (int set = 0; set < (1 << n); ++set) {
    // for all subsets of that set
    for (int subset = set; subset; subset = (subset - 1) & set) {
        // do something with the subset
    }
}

// Alternatively - also can replace (1 << n) with pow(2, n)
for (int i = 0; i < (1 << n); ++i) {
    for (int j = 0; j < n; ++j) {
        if ((i >> j) & 1) {
            // do something with A[j]
        }
    }
}
```

3.3 Travelling Sales Person

```
const int N = 20;
const int INF = 1e9;
int n, adj[N][N];   // assume this is given.
int dp[1 << N][N];  // dp[x][i] is the shortest 0->i path visiting set bits of x

int tsp(void) {
    for (int mask = 0; mask < (1 << n); mask++)
        for (int city = 0; city < n; city++) dp[mask][city] = INF;
```

```
    dp[1][0] = 0;  // 1 represents seen set {0}

    int ans = INF;
    for (int mask = 1; mask < (1 << n); mask++)  // for every subset of cities seen so far
        for (int cur = 0; cur < n; cur++)
            if (mask & (1 << cur)) {        // cur must be one of the cities seen so far
                int cdp = dp[mask][cur];    // distance travelled so far
                if (mask == (1 << n) - 1)  // seen all cities, return to 0
                    // unlike the traditional TSP, we don't have to add adj[cur][0]
                    // to account for an edge back to vertex 0
                    ans = min(ans, cdp);
                for (int nxt = 0; nxt < n; nxt++)
                    if (!(mask & (1 << nxt)))  // try going to a new city
                        // new seen set is mask union {nxt}, and we will be at nxt
                        // distance incurred to get to this state is now no worse than
                        // cdp (current distance incurred) + edge from cur to nxt
                        dp[mask | (1 << nxt)][nxt] =
                            min(dp[mask | (1 << nxt)][nxt], cdp + adj[cur][nxt]);
            }
    return ans;
}
```

4 Graph Algorithms

4.1 Breath First Search

```
vector<int> edges[N];
int dist[N];
int prev[N];

void bfs(int start) {
    fill(dist, dist + N, -1);
    dist[start] = 0;
    prev[start] = -1;

    queue<int> q;
    q.push(start);
    while (!q.empty()) {
        int c = q.front();
        q.pop();
        for (int nxt : edges[c]) {
            if (dist[nxt] == -1) {
                dist[nxt] = dist[c] + 1;
                prev[nxt] = c;
                q.push(nxt);
            }
        }
    }
}
```

4.2 Depth First Search

```
bool seen[N];

void dfs(int u) {
    if (seen[u]) return;
    seen[u] = true;
    for (int v : edges[u]) dfs(v);
}
```

4.3 Bridge Finding

```
vector<int> edges[N];
int preorder[N];  // initialise to -1
int T = 0;
int reach[N];
vector<pair<int, int>> bridges;

void dfs(int u, int from = -1) {
    preorder[u] = T++;
    reach[u] = preorder[u];

    for (int v : edges[u])
        if (v != from) {
            if (preorder[v] == -1) {
                dfs(v, u);
                if (reach[v] == preorder[v]) bridges.emplace_back(u, v);
            }
            reach[u] = min(reach[u], reach[v]);
        }
}
```

4.4 Directed Cycle Detection

```
vector<int> edges[N];
int seen[N];
int active[N];

bool has_cycle(int u) {
    if (seen[u]) return false;
    seen[u] = true;
    active[u] = true;
    for (int v : edges[u]) {
        if (active[v] || has_cycle(v)) return true;
    }
    active[u] = false;
    return false;
}
```

4.5 Tree Representation

```
const int N = 1e6 + 5;

vector<int> edges[N];

int par[N];                // Parent. -1 for the root.
vector<int> children[N];   // Your children in the tree.
int size[N];               // As an example: size of each subtree.

void constructTree(int c, int cPar = -1) {
    par[c] = cPar;
    size[c] = 1;
    for (int nxt : edges[c]) {
        if (nxt == par[c]) continue;
        constructTree(nxt, c);
        children[c].push_back(nxt);
        size[c] += size[nxt];
    }
}
```

4.6 Binary Lifting

```
const int N = 200010;
const int D = 30;  // ceil(log2(10^9))
int parent[N][D];

void precomp() {
    for (int i = 1; i <= n; ++i) cin >> parent[i][0];

    for (int j = 1; j < D; ++j) {
        for (int i = 1; i <= n; ++i) parent[i][j] = parent[parent[i][j - 1]][j - 1];
    }
}

int kth_parent(int x, int k) {
    for (int j = 0; j < D; ++j) {
        if (k & (1 << j)) x = parent[x][j];
    }
}
```

4.7 SCC's

```
template <class G, class F>
int dfs(int j, G& g, F& f) {
    int low = val[j] = ++Time, x;
    , z.push_pack(j);
    for (auto e : g[j])
        if (comp[e] < 0) low = min(low, val[e] ?: dfs(e, g, f));

    if (low == val[j]) {
        do {
            x = z.back();
            z.pop_back();
            comp[x] = ncomps;
        } while (x != j);
        f(cont);
        cont.clear();
        ++ncomps;
    }
    return val[j] = low;
}

template <class G, class F>
void scc(G& g, F f) {
    int n = g.size();
    val.assign(n, 0);
    comp.assign(n, -1);
    Time = ncomps = 0;
    for (int i = 0; i < n; ++i)
        if (comp[i] < 0) dfs(i, g, f);
}
```

4.8 Topological Sort

```
vector<int> topSort(const vector<vector<int>>& g) {
    vector<int> indeg(g.size()), q;

    for (auto& li : g) {
        for (int x : li) ++indeg[x];
    }
```

```

    for (int i = 0; i < g.size(); ++i) {
        if (indeg[i] == 0) q.push_back(i);
    }

    for (int j = 0; j < q.size(); ++j) {
        for (int x : g[q[j]]) {
            if (--indeg[x] == 0) q.push_back(x);
        }
    }

    return q;
}
```

4.9 Compute SCC DAG

```
int main() {
    cin >> n >> m;

    for (int i = 0; i < m; ++i) {
        int a, b;
        cin >> a >> b;
        edges[a].push_back(b);
        edges_r[b].push_back(a);
    }

    int nsccs = compute_sccs();
    for (int i = 1; i <= n; ++i) {
        for (int j : edges[i]) {
            if (scc[i] != scc[j]) dag[scc[i]].insert(scc[j]);
        }
    }

    vector<int> topo = topsort();
}
```

4.10 2-SAT

```
struct TwoSatSolver {
    int n_vars;
    int n_vertices;
    vector<vector<int>> adj, adj_t;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;

    TwoSatSolver(int _n_vars)
        : n_vars(_n_vars),
          n_vertices(2 * n_vars),
          adj(n_vertices),
          adj_t(n_vertices),
          used(n_vertices),
          order(),
          comp(n_vertices, -1),
          assignment(n_vars) {
        order.reserve(n_vertices);
    }
    void dfs1(int v) {
        used[v] = true;
        for (int u : adj[v]) {
```

```
            if (!used[u]) dfs1(u);
        }
        order.push_back(v);
    }

    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : adj_t[v]) {
            if (comp[u] == -1) dfs2(u, cl);
        }
    }

    bool solve_2SAT() {
        order.clear();
        used.assign(n_vertices, false);
        for (int i = 0; i < n_vertices; ++i) {
            if (!used[i]) dfs1(i);
        }

        comp.assign(n_vertices, -1);
        for (int i = 0, j = 0; i < n_vertices; ++i) {
            int v = order[n_vertices - i - 1];
            if (comp[v] == -1) dfs2(v, j++);
        }

        assignment.assign(n_vars, false);
        for (int i = 0; i < n_vertices; i += 2) {
            if (comp[i] == comp[i + 1]) return false;
            assignment[i / 2] = comp[i] > comp[i + 1];
        }
        return true;
    }

    void add_disjunction(int a, bool na, int b, bool nb) {
        // na and nb signify whether a and b are to be negated
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].push_back(b);
        adj[neg_b].push_back(a);
        adj_t[b].push_back(neg_a);
        adj_t[a].push_back(neg_b);
    }

    static void example_usage() {
        TwoSatSolver solver(3);                         // a, b, c
        solver.add_disjunction(0, false, 1, true);    //     a  v  not b
        solver.add_disjunction(0, true, 1, true);     // not a  v  not b
        solver.add_disjunction(1, false, 2, false);   //     b  v      c
        solver.add_disjunction(0, false, 0, false);   //     a  v      a
        assert(solver.solve_2SAT() == true);
        auto expected = vector<bool>(True, False, True);
        assert(solver.assignment == expected);
    }
};
```

4.11 Kruskal's Algorithm

```
struct edge {
    int u, v, w;
};
```

```
bool operator<(const edge& a, const edge& b) { return a.w < b.w; }

edge edges[N];
int root(int u);           // union-find root with path compression
void join(int u, int v);  // union-find join with size heuristic

int mst() {
    sort(edges, edges + m);  // sort by increasing weight
    int total_weight = 0;
    for (int i = 0; i < m; i++) {
        edge& e = edges[i];
        if (root(e.u) != root(e.v)) {
            total_weight += e.w;
            join(e.u, e.v);
        }
    }
    return total_weight;
}
```

4.12 Prim's Algorithm

```
typedef pair<int, int> ii;

vector<ii> edges[N];  // pairs of (weight, v)
bool in_tree[N];
priority_queue<ii, vector<ii>, greater<ii>> pq;

int mst() {
    int total_weight = 0;
    in_tree[0] = true;
    for (auto edge : edges[0]) pq.emplace(edge.first, edge.second);
    while (!pq.empty()) {
        auto edge = pq.top();
        pq.pop();
        if (in_tree[edge.second]) continue;
        in_tree[edge.second] = true;
        total_weight += edge.first;
        for (auto edge : edges[edge.second]) pq.emplace(edge.first, edge.second);
    }
    return total_weight;
}
```

4.13 Shortest Path Algorithms

4.13.1 Dijkstra's Algorithm

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef pair<ll, int> edge;  // (distance, vertex)
const int N = 100100;

vector<edge> edges[N];
ll dist[N];
bool seen[N];
priority_queue<edge, vector<edge>, greater<edge>> pq;

void dijkstra(int s) {
    fill(seen, seen + N, false);
```

```
    pq.push(edge(0, s));
    while (!pq.empty()) {
        edge cur = pq.top();
        pq.pop();
        int v = cur.second;
        ll d = cur.first;
        if (seen[v]) continue;

        dist[v] = d;
        seen[v] = true;

        for (edge nxt : edges[v]) {
            int u = nxt.second;
            ll weight = nxt.first;
            if (!seen[u]) pq.push(edge(d + weight, u));
        }
    }
}
```

4.13.2 Bellman Ford

```
const ll INF = LLONG_MAX;
struct Edge {
    int a, b, w;
    int s() { return a < b ? a : -a; }
};

struct Node {
    ll dist = INF;
    int prev = -1;
};

void bellmanFord(vector<Node>& nodes, vector<Edge>& edges, int s) {
    nodes[s].dist = 0;
    sort(edges.begin(), edges.end(), [](Edge a, Edge b) { return a.s() < b.s(); });

    int lim = nodes.size() / 2 + 2;
    for (int i = 0; i < lim; ++i) {
        for (Edge e : edges) {
            Node cur = nodes[e.a], &dest = nodes[e.b];
            if (abs(cur.dist) == INF) continue;
            ll d = cur.dist + e.w;
            if (d < dest.dist) {
                dest.prev = e.a;
                dest.dist = (i < lim - 1 ? d : -INF);
            }
        }
    }

    for (int i = 0; i < lim; ++i) {
        for (Edge e : edges) {
            if (nodes[e.a].dist == -INF) nodes[e.b].dist = -INF;
        }
    }
}
```

4.13.3 Finding Negative Cycles

```
int main() {
    cin >> n >> m;
```

```
    for (int i = 0; i < m; ++i) {
        int a, b, c;
        cin >> a >> b >> c;
        edges.push_back({a, b, c});
    }

    dist.resize(n);
    parent.resize(n);

    bool res = false;
    for (int i = 0; i < n; ++i) {
        if (visited.find(i) == visited.end() && bellman_ford((i))) {
            res = true;
            break;
        }
    }

    if (!res) cout << "NO\n";
    else {
        cout << "YES\n";

        for (int i = 0; i < n; ++i) cycleStart = parent[cycleStart];

        vector<int> cycle;
        for (int v = cycleStart;; v = parent[v]) {
            cycle.push_back(v);
            if (v == cycleStart && cycle.size() > 1) break;
        }

        reverse(cycle.begin(), cycle.end());
        for (int v : cycle) cout << v << ' ';
    }
}
```

4.13.4 Floyd Warshall

```
const ll INF = 1LL << 62;

void floydWarshall(vector<vector<ll>>& m) {
    int n = m.size();
    for (int i = 0; i < n; ++i) m[i][i] = min(m[i][i], 0LL);
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (m[i][k] != INF && m[k][j] != INF) {
                    auto newDist = max(m[i][k] + m[k][j], -INF);
                    m[i][j] = min(m[i][j], newDist);
                }
            }
        }
    }

    for (int k = 0; k < n; ++k) {
        if (m[k][k] < 0) {
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    if (m[i][k] != INF && m[k][j] != INF) m[i][j] = -INF;
                }
            }
        }
    }
}
```

5 Flow Networks

5.1 Dinic's Algorithm

```
struct Dinic {
    struct Edge {
        int to, rev;
        ll c, oc;
        Edge(int to, int rev, ll c, ll oc) : to(to), rev(rev), c(c), oc(oc) {}
        ll flow() { return max(oc - c, 0LL); }  // if you need flows
    };
    vector<int> lvl, ptr, q;
    vector<vector<Edge>> adj;

    Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}

    void addEdge(int a, int b, ll c, ll rcap = 0) {
        adj[a].push_back({b, adj[b].size(), c, c});
        adj[b].push_back({a, adj[a].size() - 1, rcap, rcap});
    }

    ll dfs(int v, int t, ll f) {
        if (v == t || !f) return f;
        for (int& i = ptr[v]; i < adj[v].size(); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.c))) {
                    e.c -= p, adj[e.to][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }

    ll calc(int s, int t) {
        ll flow = 0;
        q[0] = s;
        for (int L = 0; L < 31; ++L) do {  // 'int L=30' maybe faster for random data
                lvl = ptr = vector<int>(q.size());
                int qi = 0, qe = lvl[s] = 1;
                while (qi < qe && !lvl[t]) {
                    int v = q[qi++];
                    for (Edge e : adj[v])
                        if (!lvl[e.to] && e.c >> (30 - L)) q[qe++] = e.to, lvl[e.to] = lvl[v] +
 1;
                }
                while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
            } while (lvl[t]);
        return flow;
    }

    bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

5.2 Min-cut

```
void check_reach(int u, vector<bool>& seen) {
    if (seen[u]) return;
    seen[u] = true;
    for (int v : adjList[u])
        if (adjMat[u][v] > 0) check_reach(v, seen);
}
```

```

vector<pair<int, int>> min_cut(int s, int t) {
    ll value = dinic(s, t);

    vector<bool> seen(n, false);
    check_reach(s, seen);

    vector<pair<int, int>> ans;
    for (int u = 0; u < n; u++) {
        if (!seen[u]) continue;
        for (int v : adjList[u])
            if (!seen[v] && !is_virtual[u][v])  // need to record this in add_edge()
                ans.emplace_back(u, v);
    }
    return ans;
}
```

6 Mathematics

6.1 Fast Exponentiation

```
const ll mod = 1000000007;

ll modpow(ll b, ll e) {
    ll ans = 1;
    for (; e; b = b * b % MOD, e /= 2)
        if (e & 1) ans = ans * b % mod;
    return ans;
}
```

6.2 Mod Multiplication

```
typedef unsigned long long ull;

ull modmul(ull a, ull b, ull M) {
    ll ret = a * b - M * ull(1.L / M * a * b);
    return ret + M * (ret < 0) - M * (ret >= (ll)M);
}

ull modpow(ull b, ull e, ull mod) {
    ull ans = 1;
    for (; e; b = modmul(b, b, mod), e /= 2)
        if (e & 1) ans = modmul(ans, b, mod);
    return ans;
}
```

6.3 Miller Rabin - Primality Testing

```
bool isPrime(ull n) {
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022}, s = __builtin_ctzll(n - 1),
        d = n >> s;
    for (ull a : A) {  // ^ count trailing zeroes
        ull p = modpow(a % n, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--) p = modmul(p, p, n);
        if (p != n - 1 && i != s) return 0;
    }
    return 1;
```

```
}
```

6.4 Prime Factorisation

```
ull pollard(ull n) {
    ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    auto f = [&](ull x) { return modmul(x, x, n) + i; };
    while (t++ % 40 || gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x);
        if ((q = modmul(prd, max(x, y) - min(x, y), n))) prd = q;
        x = f(x), y = f(f(y));
    }
    return gcd(prd, n);
}

vector<ull> factor(ull n) {
    if (n == 1) return {};
    if (isPrime(n)) return {n};
    ull x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

6.5 Sieve of Eratosthenes

```
const int LIM = 1e6;
bitset<LIM> isPrime;

vector<int> sieve() {
    const int S = (int)round(sqrt(LIM)), R = LIM / 2;
    vector<int> pr = {2}, sieve(S + 1);
    pr.reserve(int(LIM / log(LIM) * 1.1));
    vector<pair<int, int>> cp;

    for (int i = 3; i <= S; i += 2)
        if (!sieve[i]) {
            cp.push_back({i, i * i / 2});
            for (int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
        }

    for (int L = 1; L <= R; L += S) {
        array<bool, S> block{};
        for (auto &[p, idx] : cp)
            for (int i = indx; i < S + L; idx = (i += p)) block[i - L] = 1;
        for (int i = 0; i < min(S, R - L); ++i)
            if (!block[i]) pr.push_back((L + i) * 2 + 1);
    }

    for (int i : pr) isPrime[i] = 1;
    return pr;
}
```

6.6 GCD

```
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

6.7 LCM

```
int lcm(int a, int b) { return a * b / gcd(a, b); }
```

6.8 Extended Euclidean Algorithm

```
ll euclid(ll a, ll b, ll &x, ll &y) {
    if (!b) return x = 1, y = 0, a;
    ll d = euclid(b, a % b, y, x);
    return y -= a / b * x, d;
}
```

6.9 Chinese Remainder Theorem

```
ll crt(ll a, ll m, ll b, ll n) {
    if (n > m) swap(a, b), swap(m, n);
    ll x, y, g = euclid(m, n, x, y);
    assert((a - b) % g == 0);  // else no solution
    x = (b - a) % n * x % n / g * m + a;
    return x < 0 ? x + m * n / g : x;
}
```

6.10 Euler's Totient Function

```
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
    for (int i = 0; i < LIM; ++i) phi[i] = i & 1 ? i : i / 2;
    for (int i = 3; i < LIM; i += 2) {
        if (phi[i] == i) {
            for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
        }
    }
}
```

6.11 Matrices

```
struct Matrix {
    int n;
    vector<vector<long long>> v;

    Matrix(int _n) : n(_n) {
        v.resize(n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) v[i].push_back(0);
    }

    Matrix operator*(const Matrix &o) const {
        Matrix res(n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++) res.v[i][j] += v[i][k] * o.v[k][j];
        return res;
    }

```

```
    static Matrix getIdentity(int n) {
        Matrix res(n);
        for (int i = 0; i < n; i++) res.v[i][i] = 1;
        return res;
    }

    Matrix operator^(long long k) const {
        Matrix res = Matrix::getIdentity(n);
        Matrix a = *this;
        while (k) {
            if (k & 1) res = res * a;
            a = a * a;
            k /= 2;
        }
        return res;
    }
};
```

6.12 Combinations

```
typedef long long ll;

const int N = 1001001;
const int MOD = 1e9 + 7;
ll f[N + 1];
ll inv[N + 1];
ll modpow(ll a, ll b, int c);  // as earlier

ll choose(ll n, ll r) { return ((f[n] * inv[r]) % MOD * inv[n - r]) % MOD; }

int main() {
    f[0] = 1;
    for (int i = 1; i < N; i++) f[i] = (i * f[i - 1]) % MOD;

    inv[N] = modpow(f[N], MOD - 2, MOD);
    for (int i = N; i >= 1; --i) inv[i - 1] = (inv[i] * i) % MOD;
}
```

7 Computational Geometry

7.1 Cross Product

```
const double EPS = 1e-8;
typedef pair<double, double> pt;
#define x first
#define y second

pt operator-(pt a, pt b) { return pt(a.x - b.x, a.y - b.y); }

bool zero(double x) { return fabs(x) <= EPS; }

double cross(pt a, pt b) { return a.x * b.y - a.y * b.x; }

// true if left or straight
// sometimes useful to instead return an int
// -1, 0 or 1: the sign of the cross product
bool ccw(pt a, pt b, pt c) { return cross(b - a, c - a) >= -EPS; }
```

7.2 Three Points Collinear

```
bool collinear(pair<ll, ll> a, pair<ll, ll> b, pair<ll, ll> c) {
    return (b.second - a.second) * (c.first - b.first) ==
           (c.second - b.second) * (b.first - a.first);
}
```

7.3 Segment-Segment Intersection

```
typedef pair<pt, pt> seg;

bool collinear(seg ab, seg cd) {  // all four points collinear
    pt a = ab.first, b = ab.second, c = cd.first, d = cd.second;
    return zero(cross(b - a, c - a)) && zero(cross(b - a, d - a));
}

double sq(double t) { return t * t; }

double dist(pt p, pt q) { return sqrt(sq(p.x - q.x) + sq(p.y - q.y)); }

bool intersect(seg ab, seg cd) {
    pt a = ab.first, b = ab.second, c = cd.first, d = cd.second;

    if (collinear(ab, cd)) {
        double maxDist =
            max({dist(a, b), dist(a, c), dist(a, d), dist(b, c), dist(b, d), dist(c, d)});
        return maxDist < dist(a, b) + dist(c, d) + EPS;
    }

    return ccw(a, b, c) != ccw(a, b, d) && ccw(c, d, a) != ccw(c, d, b);
}
```

7.4 Polygon Area (Trapezoidal Rule)

```
double area(vector<pt> pts) {
    double res = 0;
    int n = pts.size();
    for (int i = 0; i < n; i++) {
        res += (pts[i].y + pts[(i + 1) % n].y) * (pts[(i + 1) % n].x - pts[i].x);
    }
    return res / 2.0;
}
```

7.5 Polygon Area (Cross Product)

```
double area(vector<pt> pts) {
    double res = 0;
    int n = pts.size();
    for (int i = 1; i < n - 1; i++) {
        // i = 0 and i = n-1 are degenerate triangles, OK to omit
        // e.g. if i = 1 is ABC, and i = 2 is ACD, then i = 0 is AAB
        res += cross(pts[i] - pts[0], pts[i + 1] - pts[0]);
    }
    return res / 2.0;
}
```

7.6 Convex Hull

```
vector<pt> half_hull(vector<pt> pts) {
    vector<pt> res;
    for (int i = 0; i < pts.size(); i++) {
        // ccw means we have a left turn; we don't want that
        while (res.size() >= 2 && ccw(pts[i], res[res.size() - 1], res[res.size() - 2])) {
            res.pop_back();
        }
        res.push_back(pts[i]);
    }
    return res;
}

vector<pt> convex_hull(vector<pt> pts) {
    sort(pts.begin(), pts.end());
    vector<pt> top = half_hull(pts);

    reverse(pts.begin(), pts.end());
    vector<pt> bottom = half_hull(pts);

    top.pop_back();
    bottom.pop_back();
    vector<pt> res(top.begin(), top.end());
    res.insert(res.end(), bottom.begin(), bottom.end());
    return res;
}
```

7.7 Half Plane Intersection

```
typedef pair<double, double> pt;

struct line {
    double a, b, c;
};

struct half_plane {
    line l;
    bool neg;  // is the inequality <= or >=
};

const double EPS = 1e-8;

pt intersect(line f, line g) {
    double d = f.a * g.b - f.b * g.a;
    double y = (f.a * g.c - f.c * g.a) / (f.b * g.a - f.a * g.b);
    double x = (f.c * g.b - f.b * g.c) / (f.b * g.a - f.a * g.b);
    return pt(x, y);
}

bool in_half_plane(half_plane hp, pt q) {
    if (hp.neg) return hp.l.a * q.x + hp.l.b * q.y + hp.l.c <= EPS;
    else return hp.l.a * q.x + hp.l.b * q.y + hp.l.c >= -EPS;
}

vector<pt> intersect_half_planes(vector<half_plane> half_planes) {
    int n = half_planes.size();
    vector<pt> pts;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            pt p = intersect(half_planes[i].l, half_planes[j].l);
            bool fail = false;
```

```
            for (int k = 0; k < n; k++)
                if (!in_half_plane(half_planes[k], p)) fail = true;
            if (!fail) pts.push_back(p);
        }
    }

    vector<pt> res = pts;
    if (pts.size() > 2) pts = convex_hull(res);
    return pts;
}
```