Competitive Programming Algorithms

Contents

1 Introduction and Miscellaneous **3**
1.1 Template . . . 3
1.2 Codeforces Template . . . 3
1.3 Generate Files . . . 4
1.4 Binary Search . . . 4
1.5 Base 10 to Base m . . . 4
1.6 Coordinate Compression . . . 4

2 Data Structures **4**
2.1 Order Statistic Tree (Set) . . . 4
2.2 Order Statistic Tree (Map) . . . 5
2.3 Union Find . . . 5
2.4 Sparse Table . . . 5
2.5 Range Tree . . . 5
2.6 Range Tree on Trees . . . 6

3 Dynamic Programming **6**
3.1 Knapsack . . . 6
3.2 Bitsets . . . 7
3.3 Travelling Sales Person . . . 7

4 Graph Algorithms **8**
4.1 Breath First Search . . . 8
4.2 Depth First Search . . . 8
4.3 Bridge Finding . . . 8
4.4 Directed Cycle Detection . . . 9
4.5 Tree Representation . . . 9
4.6 Binary Lifting . . . 9
4.7 Kosaraju's Algorithm . . . 10
4.8 Topological Sort . . . 10
4.9 Compute SCC DAG . . . 10
4.10 2-SAT . . . 11
4.11 Kruskal's Algorithm . . . 12
4.12 Prim's Algorithm . . . 12
4.13 Shortest Path Algorithms . . . 13
4.13.1 Dijkstra's Algorithm . . . 13
4.13.2 Bellman Ford . . . 13
4.13.3 Finding Negative Cycles . . . 14
4.13.4 Floyd Warshall . . . 15

5 Flow Networks **15**
5.1 Dinic's Algorithm . . . 15
5.2 Min-cut . . . 16

6 Mathematics **17**
- 6.1 Fast Exponentiation . . . 17
- 6.2 Primality Testing . . . 17
- 6.3 Prime Factorisation . . . 17
- 6.4 Sieve of Eratosthenes . . . 17
- 6.5 GCD . . . 18
- 6.6 LCM . . . 18
- 6.7 Extended Euclidean Algorithm . . . 18
- 6.8 Matrices . . . 18
- 6.9 Combinations . . . 19

7 Computational Geometry **19**
- 7.1 Cross Product . . . 19
- 7.2 Three Points Collinear . . . 19
- 7.3 Segment-Segment Intersection . . . 20
- 7.4 Polygon Area (Trapezoidal Rule) . . . 20
- 7.5 Polygon Area (Cross Product) . . . 20
- 7.6 Convex Hull . . . 21
- 7.7 Half Plane Intersection . . . 21

1 Introduction and Miscellaneous

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)

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 Codeforces Template

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

typedef long long ll;

/* =========================================== */

void solve() {
    int n;
    cin >> n;
}

/* =========================================== */

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

    int t = 1;
    cin >> t;
    while (t--) solve();
}
```

1.3 Generate Files

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

1.4 Binary Search

```
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

```
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;

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
    ordered_set;
```

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;

typedef tree<int, char, less<int>, rb_tree_tag, tree_order_statistics_node_update> ordered_map;
```

2.3 Union Find

```
const int N = 200010;
int parent[N];
int subtree_size[N];

void init(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        subtree_size[i] = 1;
    }
}

int root(int x) { return parent[x] == x ? x : parent[x] = root(parent[x]); }

void join(int x, int y) {
    x = root(x);
    y = root(y);
    if (x == y) return;
    if (subtree_size[y] < subtree_size[x]) swap(x, y);
    parent[x] = y;
    subtree_size[y] += subtree_size[x];
}
```

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 Range Tree

```cpp
const int N = 100100;
int tree[1 << 18];  // 2^17 = 131,072

int n;

int query(int qL, int qR, int i = 1, int cL = 0, int cR = n) {
    if (cL == qL && cR == qR) return tree[i];
    int mid = (cL + cR) / 2;
    int ans = 0;
    if (qL < mid) ans += query(qL, min(qR, mid), i * 2, cL, mid);
    if (qR > mid) ans += query(max(qL, mid), qR, i * 2 + 1, mid, cR);
    return ans;
}

void update(int p, int v, int i = 1, int cL = 0, int cR = n) {
    if (cR - cL == 1) {
        tree[i] = v;
        return;
    }

    int mid = (cL + cR) / 2;
    if (p < mid)
        update(p, v, i * 2, cL, mid);
    else
        update(p, v, i * 2 + 1, mid, cR);
    tree[i] = tree[i * 2] + tree[i * 2 + 1];
}

void debug(int i = 1, int cL = 0, int cR = n) {
    cerr << "tree[" << cL << "," << cR << ") = " << tree[i];

    if (cR - cL > 1) {
        int mid = (cL + cR) / 2;
        debug(i * 2, cL, mid);
        debug(i * 2 + 1, mid, cR);
    }
}
```

2.6 Range Tree on Trees

```cpp
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(int id, int v) { update(indexInRangeTree[id], v); }
long long query_subtree(int id) { return query(startRange[id], endRange[id]); }
```

3 Dynamic Programming

3.1 Knapsack

```cpp
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

```cpp
// 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

```cpp
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 Kosaraju's Algorithm

```
int scc[N];
vector<int> edges[N], edges_r[N];
int n, m;

bool seen[N], seen_r[N];
int postorder[N];
int p = 0;

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

void dfs_r(int u, int mark) {
    if (seen_r[u]) return;
    seen_r[u] = true;
    scc[u] = mark;
    for (int v : edges_r[u]) dfs_r(v, mark);
}

int compute_sccs() {
    int sccs = 0;
    for (int i = 1; i <= n; i++)
        if (!seen[i]) dfs(i);

    for (int i = p - 1; i >= 0; i--) {
        int u = postorder[i];
        if (!seen_r[u]) dfs_r(u, sccs++);
    }
    return sccs;
}
```

4.8 Topological Sort

```
set<int> dag[N];  // edges
bool seen_dag[N];

void compute_topsort(int u, vector<int>& postorder) {
    if (seen_dag[u]) return;
    seen_dag[u] = true;
    for (int v : dag[u]) compute_topsort(v, postorder);
    postorder.push_back(u);
}

vector<int> topsort() {
    vector<int> res;
    for (int i = 0; i < nsccs; i++) compute_topsort(i, res);
    reverse(res.begin(), res.end());
    return res;
}
```

4.9 Compute SCC DAG

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

```

```cpp
    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

```cpp
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);
    }
};
```

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 long long INF = 1e9 + 7;

struct edge {
    int u, v, w;  // u -> v of weight w
    edge(int _u, int _v, int _w) : u(_u), v(_v), w(_w) {}
};

int n, m, cycleStart;
vector<long long> dist;
```

```cpp
vector<int> parent;
vector<edge> edges;
set<int> visited;

bool relax() {
    bool relaxed = false;
    for (edge e : edges) {
        if (dist[e.u] != INF && dist[e.v] > dist[e.u] + e.w) {
            relaxed = true;
            dist[e.v] = dist[e.u] + e.w;
            parent[e.v] = e.u;
            cycleStart = e.v;
            visited.insert(e.u);
        }
    }
    return relaxed;
}

bool bellman_ford(int start) {
    fill(dist.begin(), dist.end(), INF);
    fill(parent.begin(), parent.end(), -1);
    dist[start] = 0;
    for (int i = 0; i < n - 1; i++)
        if (!relax()) break;

    return relax();
}
```

4.13.3 Finding Negative Cycles

```cpp
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

```
for (int u = 0; u < n; ++u)
    for (int v = 0; v < n; ++v) dist[u][v] = INF;

for (edge e : edges) dist[e.u][e.v] = e.w;

for (int u = 0; u < n; ++u) dist[u][u] = 0;

for (int i = 0; i < n; i++)
    for (int u = 0; u < n; u++)
        for (int v = 0; v < n; v++) dist[u][v] = min(dist[u][v], dist[u][i] + dist[i][v]);
```

5 Flow Networks

5.1 Dinic's Algorithm

```
typedef long long ll;

const int INF = 1e9 + 7;

struct FlowNetwork {
    int n;
    vector<vector<ll>> adjMat, adjList;
    // level[v] stores dist from s to v
    // uptochild[v] stores first non-useless child.
    vector<int> level, uptochild;

    FlowNetwork(int _n) : n(_n) {
        // adjacency matrix is zero-initialised
        adjMat.resize(n);
        for (int i = 0; i < n; i++) adjMat[i].resize(n);
        adjList.resize(n);
        level.resize(n);
        uptochild.resize(n);
    }

    void add_edge(int u, int v, ll c) {
        // add to any existing edge without overwriting
        adjMat[u][v] += c;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    void flow_edge(int u, int v, ll c) {
        adjMat[u][v] -= c;
        adjMat[v][u] += c;
    }

    // constructs the level graph and returns whether the sink is still reachable
    bool bfs(int s, int t) {
        fill(level.begin(), level.end(), -1);
        queue<int> q;
        q.push(s);
        level[s] = 0;
        while (!q.empty()) {
```

```cpp
            int u = q.front();
            q.pop();
            uptochild[u] = 0;  // reset uptochild
            for (int v : adjList[u])
                if (adjMat[u][v] > 0) {
                    if (level[v] != -1)  // already seen
                        continue;
                    level[v] = level[u] + 1;
                    q.push(v);
                }
        }
        return level[t] != -1;
    }

    // finds an augmenting path with up to f flow.
    ll augment(int u, int t, ll f) {
        if (u == t) return f;  // base case.
        // note the reference here! we increment uptochild[u], i.e. walk through u's neighbours
        // until we find a child that we can flow through
        for (int& i = uptochild[u]; i < adjList[u].size(); i++) {
            int v = adjList[u][i];
            if (adjMat[u][v] > 0) {
                // ignore edges not in the BFS tree.
                if (level[v] != level[u] + 1) continue;
                // revised flow is constrained also by this edge
                ll rf = augment(v, t, min(f, adjMat[u][v]));
                // found a child we can flow through!
                if (rf > 0) {
                    flow_edge(u, v, rf);
                    return rf;
                }
            }
        }
        level[u] = -1;
        return 0;
    }

    ll dinic(int s, int t) {
        ll res = 0;
        while (bfs(s, t))
            for (ll x = augment(s, t, INF); x; x = augment(s, t, INF)) res += x;
        return res;
    }
};
```

5.2 Min-cut

```cpp
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 int MOD = 1e9 + 7;
typedef long long ll;

ll modpow(ll x, ll n, int m) {
    if (n == 0) return 1;

    ll a = modpow(x, n / 2, m);
    a = a * a % m;
    if (n % 2 == 1) a = a * x % m;
    return a;
}
```

6.2 Primality Testing

```
bool isprime(int x) {
    if (x < 2) return false;

    for (int f = 2; f * f <= x; f++)
        if (x % f == 0) return false;

    return true;
}
```

6.3 Prime Factorisation

```
vector<int> primefactorize(int x) {
    vector<int> factors;
    for (int f = 2; f * f <= x; f++)
        while (x % f == 0) {
            factors.push_back(f);
            x /= f;
        }

    if (x != 1) factors.push_back(x);
    return factors;
}
```

6.4 Sieve of Eratosthenes

```
bool marked[N + 1];
vector<int> primefactorization[N + 1];

for (int i = 2; i <= N; i++) {
    if (!marked[i]) {
```

```cpp
        primefactorization[i].push_back(i);
        for (int j = 2 * i; j <= N; j += i) {
            marked[j] = true;
            int tmp = j;
            while (tmp % i == 0) {
                primefactorization[j].push_back(i);
                tmp /= i;
            }
        }
    }
}
```

6.5 GCD

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

6.6 LCM

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

6.7 Extended Euclidean Algorithm

```cpp
int euclidean_algorithm(int a, int b, int& x, int& y) {
    if (a == 0) {
        x = 0;
        y = 1;
        return b;
    }
    int x1, y1;
    int d = euclidean_algorithm(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

6.8 Matrices

```cpp
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;
    }
};
```

6.9 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) >= 0; }
```

7.2 Three Points Collinear

```cpp
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

```cpp
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)

```cpp
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)

```cpp
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;
}
```