

Contents

1	Algorithms	2
1.1	Geometry	2
1.1.1	Convex Hull	2
1.1.2	Geometry Functions	2
1.2	Graph	2
1.2.1	Articulations and Bridges	2
1.2.2	Bellman-Ford	3
1.2.3	Bipartite Matching	3
1.2.4	Centroid Decomposition	3
1.2.5	Dijkstra	4
1.2.6	Dinic's	4
1.2.7	Edmonds-Karp	4
1.2.8	Floyd Warshall	5
1.2.9	Ford-Fulkerson	5
1.2.10	Hopcroft-Karp	5
1.2.11	Kosaraju	5
1.2.12	Kruskal	6
1.2.13	Lowest Common Ancestor (LCA)	6
1.2.14	Prim	7
1.2.15	Tarjan	7
1.2.16	Topological Sort	7
1.3	Math	7
1.3.1	Big Integer	7
1.3.2	Binary Exponentiation	8
1.3.3	Euler Totient (ϕ)	8
1.3.4	Extended Euclidean algorithm	9
1.3.5	Fast Fourier Transform (FFT)	9
1.3.6	Legendre's Formula	9
1.3.7	Linear Diophantine Equation	9
1.3.8	Linear Recurrence	10
1.3.9	Matrix	10
1.3.10	Modular Multiplicative Inverse	10
1.3.11	Sieve of Eratosthenes	10
1.4	Paradigm	10
1.4.1	Edit Distance	10
1.4.2	Kadane	10
1.4.3	Longest Increasing Subsequence (LIS)	11
1.4.4	Longest Common Subsequence	11
1.4.5	Ternary Search	11
1.5	String	11

CONTENTS		2
	1.5.1 Knuth-Morris-Pratt (KMP)	11
	1.5.2 Z-function	11
1.6	Structure	11
	1.6.1 AVL tree	11
	1.6.2 Binary Indexed Tree (BIT)	12
	1.6.3 Binary Indexed Tree 2D (BIT2D)	12
	1.6.4 Bitmask	12
	1.6.5 Disjoint-set	13
	1.6.6 Lazy Segment Tree	13
	1.6.7 Policy Tree	13
	1.6.8 Segment Tree	13
	1.6.9 Sqrt Decomposition	14
	1.6.10 Trie	14
2	Misc	14
2.1	Environment	15
	2.1.1 Vim Config	15
	2.1.2 Template	15

1 Algorithms

1.1 Geometry

1.1.1 Convex Hull

Time: $\mathcal{O}(n \log n)$

Space: $\mathcal{O}(n)$

```
struct ConvexHull {
    using point = pair<double, double>;

    // The three points are a counter-clockwise turn if
    // cross > 0, clockwise if cross < 0, and collinear
    // if cross = 0.
    double cross(point a, point b, point c) {
        return (b.fi - a.fi) * (c.se - a.se) - \
            (b.se - a.se) * (c.fi - a.fi);
    }

    vector<int> run(const vector<point> &v) {
        int k = 0;
        vector<int> ans(v.size() * 2);

        sort(all(v), [](const point &a, const point &b) {
            return (a.fi == b.fi) ? (a.se < b.se) : (a.fi < b.fi);
        });

        // Uppermost part of convex hull
        for (int i = 0; i < v.size(); ++i) {
            while (k >= 2 && cross(v[ans[k-2]], v[ans[k-1]], v[i]) < 0)
                k--;
            ans[k++] = i;
        }

        // Lowermost part of convex hull
        for (int i = v.size() - 2, t = k + 1; i >= 0; --i) {
            while (k >= t && cross(v[ans[k-2]], v[ans[k-1]], v[i]) < 0)
                k--;
            ans[k++] = i;
        }

        ans.resize(k - 1);
        return ans;
    }
};
```

1.1.2 Geometry Functions

```
#define to_deg(x) ((x * 180.0) / M_PI)
#define to_rad(x) ((x * M_PI) / 180.0)
```

```
template <typename T>
struct Point {
    T x, y;
```

```
Point() {}
Point(T x, T y) : x(x), y(y) {}

Point operator+(Point p) { return Point(x+p.x, y+p.y); }
Point operator-(Point p) { return Point(x-p.x, y-p.y); }

T dot(Point p) { return (x*p.x) + (y*p.y); }
T cross(Point p) { return (x*p.y) - (y*p.x); }

// Returns angle between this and p:
// atan2(y, x) is in the range [-180, 180]. To
// get [0, 360], atan2(-y, -x) + 180 is used
T angle(Point p) {
    return to_deg(atan2(-cross(p), -dot(p))) + 180.0;
}

// Returns cosine value between this and p.
T cosine(Point p) {
    return (dot(p) / (sqrt(dot(*this))*sqrt(p.dot(p))));
}

// Returns sine value between this and p.
T sine(Point p) {
    return (cross(p) / (sqrt(dot(*this))*sqrt(p.dot(p))));
}

// Returns whether point is inside the triable
// abc or not.
bool inside_triangle(Point a, Point b, Point c) {
    bool c1 = (*this - b).cross(a - b) < 0;
    bool c2 = (*this - c).cross(b - c) < 0;
    bool c3 = (*this - a).cross(c - a) < 0;
    return c1 == c2 && c1 == c3;
}

// Finds orientation of ordered triplet (a,b,c).
// Colinear (0), Clockwise (1), Counterclockwise (2)
static int orientation(Point a, Point b, Point c) {
    T val = (b - a).cross(c - b);
    if (val == 0) return 0;
    return (val > 0) ? 1 : 2;
}

template <typename T>
struct Segment {
    Point<T> a, b;

    Segment(Point a, Point b) : a(a), b(b) {}

    // Checks if points p and q are on the same side
    // of the segment.
    bool same_side(Point p, Point q) {
        T cpp = (p - a).cross(b - a);
        T cpq = (q - a).cross(b - a);
        return ((cpp > 0 && cpq > 0) ||
            (cpp < 0 && cpq < 0));
    }

    // Checks if point p is on the segment.
```

```
bool on_segment(Point p) {
    return (p.x <= max(a.x, b.x) &&
        p.x >= min(a.x, b.x) &&
        p.y <= max(a.y, b.y) &&
        p.y >= min(a.y, b.y));
}

// Checks if segment intersects with s.
bool intersect(Segment s) {
    int o1 = Point::orientation(a, b, s.a);
    int o2 = Point::orientation(a, b, s.b);
    int o3 = Point::orientation(s.a, s.b, a);
    int o4 = Point::orientation(s.a, s.b, b);

    if (o1 != o2 && o3 != o4)
        return true;

    if (o1 == 0 && on_segment(s.a)) return true;
    if (o2 == 0 && on_segment(s.b)) return true;
    if (o3 == 0 && s.on_segment(a)) return true;
    if (o4 == 0 && s.on_segment(b)) return true;

    return false;
}

template <typename T>
struct Polygon {
    vector<Point<T>> v;

    Polygon() {}
    Polygon(vector<Point> v) : v(v) {}

    // Adds a vertex to the polygon.
    void add_point(Point p) { v.pb(p); }

    // Returns area of polygon (only works when vertices
    // are sorted in clockwise or counterclockwise order).
    double area() {
        double ans = 0;
        for (int i = 0; i < v.size(); ++i)
            ans += v[i].cross(v[(i + 1) % v.size()]);

        return fabs(ans) / 2.0;
    }
};
```

1.2 Graph

1.2.1 Articulations and Bridges

Time: $\mathcal{O}(V + E)$

Space: $\mathcal{O}(V + E)$

```
vector<int> graph[MAX];
```

```
struct ArticulationsBridges {
```

```

int N;
vector<int> vis, par, L, low;

vector<ii> brid;
vector<int> arti;

ArticulationsBridges(int N) :
    N(N), vis(N), par(N), L(N), low(N) {}

void init() {
    fill(all(L), 0);
    fill(all(vis), 0);
    fill(all(par), -1);
}

void dfs(int x) {
    int child = 0;
    vis[x] = 1;

    for (auto i : graph[x]) {
        if (!vis[i]) {
            child++;
            par[i] = x;

            low[i] = L[i] = L[x] + 1;
            dfs(i);
            low[x] = min(low[x], low[i]);

            if ((par[x] == -1 && child > 1) ||
                (par[x] != -1 && low[i] >= L[x]))
                arti.pb(x);

            if (low[i] > L[x])
                brid.pb(ii(x, i));
        } else if (par[x] != i)
            low[x] = min(low[x], L[i]);
    }
}

void run() {
    for (int i = 0; i < N; ++i)
        if (!vis[i])
            dfs(i);

    sort(all(arti));
    arti.erase(unique(all(arti)), arti.end());
}
};

```

1.2.2 Bellman-Ford

Time: $\mathcal{O}(V \times E)$
Space: $\mathcal{O}(V + E)$

```

struct BellmanFord {
    struct Edge { int u, v, w; };

    int N;
    vector<int> dist;
    vector<Edge> graph;

    BellmanFord(int N) :

```

```

    N(N), dist(N) {}

    void init() {
        fill(all(dist), inf);
    }

    int run(int s, int d) {
        dist[s] = 0;

        for (int i = 0; i < N; ++i)
            for (auto e : graph)
                if (dist[e.u] != inf &&
                    dist[e.u] + e.w < dist[e.v])
                    dist[e.v] = dist[e.u] + e.w;

        // Check for negative cycles, return -inf if
        // there is one
        for (auto e : graph)
            if (dist[e.u] != inf &&
                dist[e.u] + w < dist[e.v])
                return -inf;

        return dist[d];
    }
};

```

1.2.3 Bipartite Matching

Time: $\mathcal{O}(V \times E)$
Space: $\mathcal{O}(V \times E)$

```

vector<int> graph[MAX];

struct BipartiteMatching {
    int N;
    vector<int> vis, match;

    BipartiteMatching(int N) :
        N(N), vis(N), match(N) {}

    void init() {
        fill(all(vis), 0);
        fill(all(match), -1);
    }

    int dfs(int x) {
        if (vis[x])
            return 0;

        vis[x] = 1;
        for (auto i : graph[x])
            if (match[i] == -1 || dfs(match[i])) {
                match[i] = x;
                return 1;
            }

        return 0;
    }

    int run() {
        int ans = 0;
        for (int i = 0; i < N; ++i)
            ans += dfs(i);
    }
};

```

```

        return ans;
    }
};

```

1.2.4 Centroid Decomposition

Description:

The Centroid Decomposition of a tree is a tree where: 1) its root is the centroid of the original tree, and 2) its children are the centroid of each tree resulting from the removal of the root from the original tree.

The result is a tree with $\log n$ height, where the path from a to b , in the original tree, can be decomposed into the path from a to $\text{lca}(a, b)$ and from $\text{lca}(a, b)$ to b .

This is useful because each one of the n^2 paths of the original tree is a concatenation of two paths in a set of $\mathcal{O}(n \log n)$ paths (from each node to all of its ancestors in the centroid decomposition).

Time: $\mathcal{O}(V \log V)$
Space: $\mathcal{O}(V + E)$

```

// Must be a tree
vector<int> graph[MAX];

struct CentroidDecomposition {
    vector<int> par, size, marked;

    CentroidDecomposition(int N) :
        par(N), size(N), marked(N)
    { init(); }

    void init() {
        fill(all(marked), 0);
        build(0); // 0-indexed vertices
    }

    void build(int x, int p = -1) {
        int n = dfs(x);
        int centroid = get_centroid(x, n);

        marked[centroid] = 1;
        par[centroid] = p;

        for (auto i : graph[centroid])
            if (!marked[i])
                build(i, centroid);
    }

    // Calculates size of every subtree.
    int dfs(int x, int p = -1) {
        size[x] = 1;
        for (auto i : graph[x])
            if (i != p && !marked[i])
                size[x] += dfs(i, x);
        return size[x];
    }

    int get_centroid(int x, int n, int p = -1) {
        for (auto i : graph[x])
            if (i != p && size[i] > n / 2 && !marked[i])

```

```

    return get_centroid(i, x, n);
    return x;
}

int operator[](int i) {
    return par[i];
}
};

```

1.2.5 Dijkstra

Description:

Dijkstra's algorithm for finding the shortest paths between nodes in a graph. It works by greedily extending the shortest path at each step.

Doesn't work with negative-weighted edges, for that, Bellman-Ford algorithm must be used.

Time: $\mathcal{O}(E + V \log V)$

Space: $\mathcal{O}(V + E)$

```

vector<int> graph[MAX];

struct Dijkstra {
    int N;
    vector<int> dist, vis;

    Dijkstra(int N) :
        N(N), dist(N), vis(N)
    { init(); }

    void init() {
        fill(all(vis), 0);
        fill(all(dist), inf);
    }

    int run(int s, int d) {
        set<ii> pq;

        dist[s] = 0;
        pq.insert(ii(0, s));

        while (pq.size() != 0) {
            int u = pq.begin()->se;
            pq.erase(pq.begin());

            if (vis[u]) continue;
            vis[u] = 1;

            for (auto i : graph[u]) {
                if (!vis[i.fi] && dist[i.fi] > dist[u] + i.se) {
                    dist[i.fi] = dist[u] + i.se;
                    pq.insert(ii(dist[i.fi], i.fi));
                }
            }
        }

        return dist[d];
    }
};

```

1.2.6 Dinic's

Time: $\mathcal{O}(E \times V^2)$

Space: $\mathcal{O}(V + E)$

```

struct Dinic {
    struct Edge { int u, f, c, r; };

    int N;
    vector<int> depth, start;
    vector<vector<Edge>> graph;

    Dinic(int N) :
        N(N), depth(N), start(N), graph(N) {}

    void add_edge(int s, int t, int c) {
        Edge forw = { t, 0, c, (int) graph[t].size() };
        Edge back = { s, 0, 0, (int) graph[s].size() };
        graph[s].pb(forw);
        graph[t].pb(back);
    }

    bool bfs(int s, int t) {
        queue<int> Q;
        Q.push(s);

        fill(all(depth), -1);
        depth[s] = 0;

        while (!Q.empty()) {
            int v = Q.front(); Q.pop();

            for (auto i : graph[v])
                if (depth[i.u] == -1 && i.f < i.c) {
                    depth[i.u] = depth[v] + 1;
                    Q.push(i.u);
                }
        }

        return depth[t] != -1;
    }

    int dfs(int s, int t, int f) {
        if (s == t)
            return f;

        for ( ; start[s] < graph[s].size(); ++start[s]) {
            Edge &e = graph[s][start[s]];

            if (depth[e.u] == depth[s] + 1 && e.f < e.c) {
                int min_f = dfs(e.u, t, min(f, e.c - e.f));

                if (min_f > 0) {
                    e.f += min_f;
                    graph[e.u][e.r].f -= min_f;
                    return min_f;
                }
            }
        }

        return 0;
    }

    int run(int s, int t) {
        int ans = 0;

```

```

        while (bfs(s, t)) {
            fill(all(start), 0);
            while (int flow = dfs(s, t, inf))
                ans += flow;
        }

        return ans;
    }
};

```

1.2.7 Edmonds-Karp

Time: $\mathcal{O}(V \times E^2)$

Space: $\mathcal{O}(V^2)$

```

int rg[MAX][MAX];
int graph[MAX][MAX];

struct EdmondsKarp {
    int N;
    vector<int> par, vis;

    EdmondsKarp(int N) :
        N(N), par(N), vis(N)
    { init(); }

    void init() {
        fill(all(vis), 0);
    }

    bool bfs(int s, int t) {
        queue<int> Q;
        Q.push(s);
        vis[s] = true;

        while (!Q.empty()) {
            int u = Q.front(); Q.pop();

            if (u == t)
                return true;

            for (int i = 0; i < N; ++i)
                if (!vis[i] && rg[u][i]) {
                    vis[i] = true;
                    par[i] = u;
                    Q.push(i);
                }
        }

        return false;
    }

    int run(int s, int t) {
        int ans = 0;
        par[s] = -1;

        memcpy(rg, graph, sizeof(graph));

        while (bfs(s, t)) {
            int flow = inf;
            for (int i = t; par[i] != -1; i = par[i])
                flow = min(flow, rg[par[i]][i]);

```

```

    for (int i = t; par[i] != -1; i = par[i]) {
        rg[par[i]][i] -= flow;
        rg[i][par[i]] += flow;
    }

    ans += flow;
    init();
}

return ans;
}
};

```

1.2.8 Floyd Warshall

Time: $\mathcal{O}(V^3)$
Space: $\mathcal{O}(V^2)$

```

int dist[MAX][MAX];
int graph[MAX][MAX];

struct FloydWarshall {
    int N;

    FloydWarshall(int N) :
        N(N) {}

    int run() {
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                dist[i][j] = graph[i][j];

        for (int k = 0; k < N; ++k)
            for (int i = 0; i < N; ++i)
                for (int j = 0; j < N; ++j)
                    dist[i][j] = min(dist[i][j],
                                       dist[i][k] + dist[k][j]);
    }
};

```

1.2.9 Ford-Fulkerson

Time: $\mathcal{O}(E \times f)$
Space: $\mathcal{O}(V^2)$

```

int rg[MAX][MAX];
int graph[MAX][MAX];

struct FordFulkerson {
    int N;
    vector<int> par, vis;

    FordFulkerson(int N) :
        N(N), par(N), vis(N)
    { init(); }

    void init() { fill(all(vis), 0); }

    bool dfs(int s, int t) {
        vis[s] = true;

```

```

        if (s == t)
            return true;

        for (int i = 0; i < N; ++i)
            if (!vis[i] && rg[s][i]) {
                par[i] = s;

                if (dfs(i, t))
                    return true;
            }

        return false;
    }

    int run(int s, int t) {
        int ans = 0;
        par[s] = -1;

        memcpy(rg, graph, sizeof(graph));

        while (dfs(s, t)) {
            int flow = inf;
            for (int i = t; par[i] != -1; i = par[i])
                flow = min(flow, rg[par[i]][i]);

            for (int i = t; par[i] != -1; i = par[i]) {
                rg[par[i]][i] -= flow;
                rg[i][par[i]] += flow;
            }

            ans += flow;
            init();
        }

        return ans;
    }
};

```

1.2.10 Hopcroft-Karp

Time: $\mathcal{O}(E \times \sqrt{V})$
Space: $\mathcal{O}(V + E)$

```

vector<int> graph[MAX];

struct HopcroftKarp {
    int L, R;
    vector<int> dist;
    vector<int> matchL, matchR;

    HopcroftKarp(int L, int R) :
        L(L), R(R), dist(L),
        matchL(L), matchR(R)
    { init(); }

    void init() {
        fill(all(matchL), 0);
        fill(all(matchR), 0);
    }

    bool bfs() {
        queue<int> Q;

```

```

        for (int l = 1; l <= L; ++l)
            if (matchL[l] == 0) {
                dist[l] = 0;
                Q.push(l);
            } else
                dist[l] = inf;

        dist[0] = inf;
        while (!Q.empty()) {
            int l = Q.front(); Q.pop();

            if (dist[l] < dist[0])
                for (auto r : graph[l])
                    if (dist[matchR[r]] == inf) {
                        dist[matchR[r]] = dist[l] + 1;
                        Q.push(matchR[r]);
                    }

            return (dist[0] != inf);
        }

        bool dfs(int l) {
            if (l == 0)
                return true;

            for (auto r : graph[l])
                if (dist[matchR[r]] == dist[l] + 1)
                    if (dfs(matchR[r])) {
                        matchR[r] = l;
                        matchL[l] = r;
                        return true;
                    }

            dist[l] = inf;
            return false;
        }

        int run() {
            int ans = 0;

            while (bfs())
                for (int l = 1; l <= L; ++l)
                    if (matchL[l] == 0 && dfs(l))
                        ans++;

            return ans;
        }
    }
};

```

1.2.11 Kosaraju

Time: $\mathcal{O}(V + E)$
Space: $\mathcal{O}(V + E)$

```

vector<int> graph[MAX];
vector<int> transp[MAX];

struct Kosaraju {
    int N;
    stack<int> S;
    vector<int> vis;

```

```

Kosaraju(int N) :
    N(N), vis(N)
{ init(); }

void init() { fill(all(vis), 0); }

void dfs(int x) {
    vis[x] = true;

    for (auto i : transp[x])
        if (!vis[i])
            dfs(i);
}

// Fills stack with DFS starting points to find SCC.
void fill_stack(int x) {
    vis[x] = true;

    for (auto i : graph[x])
        if (!vis[i])
            fill_stack(i);

    S.push(x);
}

int run() {
    int scc = 0;

    init();
    for (int i = 0; i < N; ++i)
        if (!vis[i])
            fill_stack(i);

    // Transpose graph
    for (int i = 0; i < N; ++i)
        for (auto j : graph[i])
            transp[j].push_back(i);

    init();

    // Count SCC
    while (!S.empty()) {
        int v = S.top();
        S.pop();

        if (!vis[v]) {
            dfs(v);
            scc++;
        }
    }

    return scc;
}
};

```

1.2.12 Kruskal

Time: $\mathcal{O}(E \log V)$

Space: $\mathcal{O}(E)$

```

typedef pair<ii,int> iii;
vector<iii> edges;

```

```

struct Kruskal {
    int N;
    DisjointSet ds;

    Kruskal(int N) : N(N), ds(N) {}

    // Returns value of MST and fills mst vector with
    // the edges from the MST.
    int run(vector<iii> &mst) {

        // Sort by weight of the edges
        sort(all(edges), [&](const iii &a, const iii &b) {
            // ('>' for maximum spanning tree)
            return a.se < b.se;
        });

        int size = 0;
        for (int i = 0; i < edges.size(); i++) {
            int pu = ds.find_set(edges[i].fi.fi);
            int pv = ds.find_set(edges[i].fi.se);

            // If the sets are different, then the edge i does
            // not close a cycle
            if (pu != pv) {
                mst.pb(edges[i]);
                size += edges[i].se;
                ds.union_set(pu, pv);
            }
        }

        return size;
    }
};

```

1.2.13 Lowest Common Ancestor (LCA)

Description:

The LCA between two nodes in a tree is a node that is an ancestor to both nodes with the lowest height possible.

The algorithm works by following the path up the tree from both nodes "simultaneously" until a common node is found. The naive approach for that would be $\mathcal{O}(n)$ in the worst case. To improve that, this implementation uses "binary lifting" which is a way of figuring out the right number of up-moves needed to find the LCA by following the binary representation of the distance to the destination (similar to the "binary search by jumping"), but, for that, a preprocessing must be done to set every parent at a 2^i distance.

Time:

- preprocess: $\mathcal{O}(V \log V)$
- query: $\mathcal{O}(\log V)$

Space: $\mathcal{O}(V + E + V \log V)$

```
#define MAXLOG 20 //log2(MAX)
```

```
vector<ii> graph[MAX];
```

```

struct LCA {
    vector<int> h;

```

```

    vector<vector<int>> par, cost;

    LCA(int N) :
        h(N),
        par(N, vector<int>(MAXLOG)),
        cost(N, vector<int>(MAXLOG))
    { init(); }

    void init() {
        for (auto &i : par) fill(all(i), -1);
        for (auto &i : cost) fill(all(i), 0);
        dfs(0); // 0-indexed vertices
    }

    int op(int a, int b) {
        return a + b; // or max(a, b)
    }

    void dfs(int v, int p = -1, int c = 0) {
        par[v][0] = p;
        cost[v][0] = c;

        if (p != -1)
            h[v] = h[p] + 1;

        for (int i = 1; i < MAXLOG; ++i)
            if (par[v][i - 1] != -1) {
                par[v][i] = par[par[v][i - 1]][i - 1];
                cost[v][i] = op(cost[v][i - 1], op(cost[v][i - 1],
                    cost[par[v][i - 1]][i - 1]));
            }

        for (auto u : graph[v])
            if (p != u.fi)
                dfs(u.fi, v, u.se);
    }

    int query(int p, int q) {
        int ans = 0;

        if (h[p] < h[q])
            swap(p, q);

        for (int i = MAXLOG - 1; i >= 0; --i)
            if (par[p][i] != -1 && h[par[p][i]] >= h[q]) {
                ans = op(ans, cost[p][i]);
                p = par[p][i];
            }

        if (p == q) {
#ifdef COST
            return ans;
#else
            return p;
#endif
        }

        for (int i = MAXLOG - 1; i >= 0; --i)
            if (par[p][i] != -1 && par[p][i] != par[q][i]) {
                ans = op(ans, op(cost[p][i], cost[q][i]));
                p = par[p][i];
                q = par[q][i];
            }

#ifdef COST

```

```

    if (p == q)
        return ans;
    else
        return op(ans, op(cost[p][0], cost[q][0]));
    #else
        return par[p][0];
    #endif
}
};

```

1.2.14 Prim

Time: $\mathcal{O}(E \log E)$

Space: $\mathcal{O}(V + E)$

```

vector<ii> graph[MAX];

struct Prim {
    int N;
    vector<int> vis;

    Prim(int N) :
        N(N), vis(N)
    { init(); }

    void init() {
        fill(all(vis), 0);
    }

    int run() {
        vis[0] = true;

        priority_queue<ii> pq;
        for (auto i : graph[0])
            pq.push(ii(-i.se, -i.fi));

        int ans = 0;
        while (!pq.empty()) {
            ii front = pq.top(); pq.pop();
            int u = -front.se;
            int w = -front.fi;

            if (!vis[u]) {
                ans += w;
                vis[u] = true;

                for (auto i : graph[u])
                    if (!vis[i.fi])
                        pq.push(ii(-i.se, -i.fi));
            }
        }

        return ans;
    }
};

```

1.2.15 Tarjan

Time: $\mathcal{O}(V + E)$

Space: $\mathcal{O}(V + E)$

```

vector<int> scc[MAX];
vector<int> graph[MAX];

struct Tarjan {
    int N, ncomp, ind;

    stack<int> S;
    vector<int> vis, id, low;

    Tarjan(int N) :
        N(N), vis(N), id(N), low(N)
    { init(); }

    void init() {
        fill(all(id), -1);
        fill(all(vis), 0);
    }

    void dfs(int x) {
        id[x] = low[x] = ind++;
        vis[x] = 1;

        S.push(x);

        for (auto i : graph[x])
            if (id[i] == -1) {
                dfs(i);
                low[x] = min(low[x], low[i]);
            } else if (vis[i])
                low[x] = min(low[x], id[i]);

        // A SCC was found
        if (low[x] == id[x]) {
            int w;

            do {
                w = S.top(); S.pop();
                vis[w] = 0;
                scc[ncomp].pb(w);
            } while (w != x);

            ncomp++;
        }
    }

    int run() {
        init();
        ncomp = ind = 0;

        for (int i = 0; i < N; ++i)
            scc[i].clear();

        // Apply tarjan in every component
        for (int i = 0; i < N; ++i)
            if (id[i] == -1)
                dfs(i);

        return ncomp;
    }
};

```

1.2.16 Topological Sort

Time: $\mathcal{O}(V + E)$

Space: $\mathcal{O}(V + E)$

```

vector<int> graph[MAX];

struct TopologicalSort {
    int N;
    stack<int> S;
    vector<int> vis;

    TopologicalSort(int N) :
        N(N), vis(N)
    { init(); }

    void init() { fill(all(vis), 0); }

    bool dfs(int x) {
        vis[x] = 1;

        for (auto i : graph[x]) {
            if (vis[i] == 1) return true;
            if (!vis[i] && dfs(i)) return true;
        }

        vis[x] = 2;
        S.push(x);

        return false;
    }

    // Returns whether graph contains cycle
    // or not.
    bool run(vector<int> &tsort) {
        init();

        bool cycle = false;
        for (int i = 0; i < N; ++i)
            if (!vis[i])
                cycle |= dfs(i);

        if (cycle)
            return true;

        while (!S.empty()) {
            tsort.pb(S.top());
            S.pop();
        }

        return false;
    }
};

```

1.3 Math

1.3.1 Big Integer

Space: $\mathcal{O}(n)$

```

const int base = 1000000000;
const int base_d = 9;

```



```

struct BigInt {
    int sign = 1;
    vector<int> num;

    BigInt() {}
    BigInt(const string &x) { read(x); }

    BigInt operator+(const BigInt &x) const {
        if (sign != x.sign) return *this - (-x);

        BigInt ans = x;
        int carry = 0;
        for (int i = 0; i < max(size(), x.size()) || carry; ++i) {
            if (i == ans.size()) ans.push_back(0);

            if (i < size()) ans[i] += carry + num[i];
            else ans[i] += carry;

            carry = ans[i] >= base;
            if (carry) ans[i] -= base;
        }

        return ans;
    }

    BigInt operator-(const BigInt &x) const {
        if (sign != x.sign)
            return *this + (-x);
        if (abs() < x.abs())
            return -(x - *this);

        BigInt ans = *this;
        int carry = 0;
        for (int i = 0; i < x.size() || carry; ++i) {
            if (i < x.size()) ans[i] -= carry + x[i];
            else ans[i] -= carry;

            carry = ans[i] < 0;
            if (carry) ans[i] += base;
        }

        ans.trim();
        return ans;
    }

    // Removes leading zeros.
    void trim() {
        while (!num.empty() && num.back() == 0)
            num.pop_back();

        if (num.empty())
            sign = 1;
    }

    bool operator<(const BigInt &x) {
        if (sign != x.sign)
            return sign < x.sign;

        if (size() != x.size())
            return (size() * sign) < (x.size() * x.sign);

        for (int i = size() - 1; i >= 0; i--)
            if (num[i] != x[i])
                return (num[i] * sign) < (x[i] * x.sign);
    }
}

```

```

        return false;
    }

    bool operator==(const BigInt &x) {
        return !(*this < x) && !(x < *this);
    }

    bool operator>(const BigInt &x) const { return (x < *this); }
    bool operator<=(const BigInt &x) const { return !(x < *this); }
    bool operator>=(const BigInt &x) const { return !(*this < x); }
    bool operator!=(const BigInt &x) const { return !(*this == x); }

    // Handles -x (change of sign).
    BigInt operator-() const {
        BigInt ans = *this;
        ans.sign = -sign;
        return ans;
    }

    // Returns absolute value.
    BigInt abs() const {
        BigInt ans = *this;
        ans.sign *= ans.sign;
        return ans;
    }

    // Transforms string into BigInt.
    void read(const string &s) {
        sign = 1;
        num.clear();

        int pos = 0;
        while (pos < (int) s.size() && (s[pos] == '-' || s[pos] == '+'))
        {
            if (s[pos] == '-')
                sign = -sign;
            ++pos;
        }

        for (int i = s.size() - 1; i >= pos; i -= base_d) {
            int x = 0;
            for (int j = max(pos, i - base_d + 1); j <= i; j++)
                x = x * 10 + s[j] - '0';
            num.push_back(x);
        }

        trim();
    }

    friend istream& operator>>(istream &stream, BigInt &v) {
        string s; stream >> s;
        v.read(s);
        return stream;
    }

    friend ostream& operator<<(ostream &stream,
        const BigInt &x) {
        if (x.sign == -1)

```

```

        stream << '-';

        stream << (x.empty() ? 0 : x.back());
        for (int i = x.size() - 2; i >= 0; --i)
            stream << setw(base_d) << setfill('0') << x.num[i];

        return stream;
    }

    // Handles vector operations.
    int back() const { return num.back(); }
    bool empty() const { return num.empty(); }
    size_t size() const { return num.size(); }
    void push_back(int x) { num.push_back(x); }

    int &operator[](int i) { return num[i]; }
    int operator[](int i) const { return num[i]; }
};

```

1.3.2 Binary Exponentiation

Time: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(1)$

```

struct BinaryExponentiation {
    ll run(ll x, ll n) {
        ll ans = 1;

        while (n) {
            if (n & 1)
                ans = ans * x;

            n >>= 1;
            x = x * x;
        }

        return ans;
    }
};

```

1.3.3 Euler Totient (ϕ)

Time: $\mathcal{O}(\sqrt{n})$

Space: $\mathcal{O}(1)$

```

struct EulerTotient {
    int run(int n) {
        int result = n;

        for (int i = 2; i*i <= n; i++)
            if (n % i == 0) {
                while (n % i == 0) n /= i;
                result -= result / i;
            }

        if (n > 1)
            result -= (result / n);

        return result;
    }
};

```

1.3.4 Extended Euclidean algorithm

Time: $\mathcal{O}(\log \min(a, b))$

Space: $\mathcal{O}(1)$

```
struct ExtendedEuclidean {
    int run(int a, int b, int &x, int &y) {
        if (a == 0) {
            x = 0, y = 1;
            return b;
        }

        int x1, y1;
        int g = run(b % a, a, x1, y1);

        x = y1 - (b / a) * x1;
        y = x1;

        return g;
    }
};
```

1.3.5 Fast Fourier Transform (FFT)

Time: $\mathcal{O}(N \log N)$

Space: $\mathcal{O}(N)$

```
struct FFT {
    struct Complex {
        float r, i;

        Complex() : r(0), i(0) {}
        Complex(float r, float i) : r(r), i(i) {}

        Complex operator+(Complex b) {
            return Complex(r + b.r, i + b.i);
        }

        Complex operator-(Complex b) {
            return Complex(r - b.r, i - b.i);
        }

        Complex operator*(Complex b) {
            return Complex(r*b.r - i*b.i, r*b.i + i*b.r);
        }

        Complex operator/(Complex b) {
            float div = (b.r * b.r) + (b.i * b.i);
            return Complex((r * b.r + i * b.i) / div,
                (i * b.r - r * b.i) / div);
        }

        static inline Complex conj(Complex a) {
            return Complex(a.r, -a.i);
        }
    };

    vector<int> rev = {0, 1};
    vector<Complex> roots = {{0, 0}, {1, 0}};

    // Initializes reversed-bit vector (rev) and
    // roots of unity vector (roots)
    void init(int nbase) {
```

```
        rev.resize(1 << nbase);
        roots.resize(1 << nbase);

        // Construct rev vector
        for (int i = 0; i < (1 << nbase); ++i)
            rev[i] = (rev[i >> 1] >> 1) + \
                ((i & 1) << (nbase - 1));

        // Construct roots vector
        for (int base = 1; base < nbase; ++base) {
            float angle = 2 * M_PI / (1 << (base + 1));

            for (int i = 1 << (base - 1); i < (1 << base); ++i) {
                float angle_i = angle * (2*i + 1 - (1 << base));

                roots[i << 1] = roots[i];
                roots[(i << 1) + 1] = Complex(cos(angle_i),
                    sin(angle_i));
            }
        }

        void fft(vector<Complex> &a) {
            int n = a.size();

            for (int i = 0; i < n; ++i)
                if (i < rev[i])
                    swap(a[i], a[rev[i]]);

            for (int s = 1; s < n; s <= 1) {
                for (int k = 0; k < n; k += (s <= 1)) {
                    for (int j = 0; j < s; ++j) {
                        Complex z = a[k + j + s] * roots[j + s];
                        a[k + j + s] = a[k + j] - z;
                        a[k + j] = a[k + j] + z;
                    }
                }
            }

            vector<int> multiply(const vector<int> &a,
                const vector<int> &b)
            {
                int nbase, need = a.size() + b.size() + 1;

                for (nbase = 0; (1 << nbase) < need; ++nbase);
                init(nbase);

                int size = 1 << nbase;
                vector<Complex> fa(size);

                for (int i = 0; i < size; ++i) {
                    int x = (i < a.size() ? a[i] : 0);
                    int y = (i < b.size() ? b[i] : 0);
                    fa[i] = Complex(x, y);
                }

                fft(fa);

                Complex r(0, -0.25 / size);
                for (int i = 0; i <= (size >> 1); ++i) {
                    int j = (size - i) & (size - 1);
                    Complex z = (fa[j]*fa[j] - conj(fa[i]*fa[i])) * r;

                    if (i != j)
```

```
                        fa[j] = (fa[i]*fa[i] - conj(fa[j]*fa[j])) * r;
                    }
                }

                vector<int> res(need);
                for (int i = 0; i < need; ++i)
                    res[i] = fa[i].r + 0.5;

                return res;
            }
};
```

1.3.6 Legendre's Formula

Time: $\mathcal{O}(\log_p n)$

Space: $\mathcal{O}(1)$

```
struct Legendre {
    int run(int n, int p) {
        int x = 0;

        while (n) {
            n /= p;
            x += n;
        }

        return x;
    }
};
```

1.3.7 Linear Diophantine Equation

Description:

A Linear Diophantine Equation is an equation in the form $ax + by = c$. A solution of this equation is a pair (x, y) that satisfies the equation. The locus of (lattice) points whose coordinates x and y satisfy the equation is a straight line.

The equation has a solution only if $\gcd(a, b) | c$. In the case of existing a solution for the provided a, b, c , the infinite set of coordinates (x, y) can be obtained with $\text{get}(t)$ for $t = \dots, -2, -1, 0, 1, 2, \dots$

Time: $\mathcal{O}(\log \min(a, b))$

Space: $\mathcal{O}(1)$

```
struct Diophantine {
    int a, b, c, d;
    int x0, y0;

    bool has_solution;

    Diophantine(int a, int b, int c) :
        a(a), b(b), c(c)
    { init(); }

    void init() {
        ExtendedEuclidean ext_gcd;
```

```

int w0, z0;
d = ext_gcd.run(a, b, w0, z0);
if (c % d == 0) {
    x0 = w0 * (c / d);
    y0 = z0 * (c / d);
    has_solution = true;
} else {
    has_solution = false;
}
}

ii get(int t) {
    if (!has_solution) return ii(inf, inf);
    return ii(x0 + t * (b / d), y0 - t * (a / d));
}
};

```

1.3.8 Linear Recurrence

Time: $\mathcal{O}(\log n)$
Space: $\mathcal{O}(1)$

```

template <typename T>
matrix<T> solve(int x, int y, int n) {
    matrix<T> in(2, 2);

    // Example
    in[0][0] = x % MOD;
    in[0][1] = y % MOD;
    in[1][0] = 1;
    in[1][1] = 0;

    return fast_pow<T>(in, n);
}

```

1.3.9 Matrix

Space: $\mathcal{O}(R \times C)$

```

template <typename T>
struct Matrix {
    int r, c;
    vector<vector<T>> m;

    Matrix(int r, int c) : r(r), c(c) {
        m = vector<vector<T>>(r, vector<T>(c, 0));
    }

    Matrix operator*(Matrix a) {
        assert(r == a.c && c == a.r);

        Matrix res(r, c);
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++) {
                res[i][j] = 0;

                for (int k = 0; k < c; k++)
                    res[i][j] += m[i][k] * a[k][j];
            }
    }
};

```

```

return res;
}

vector<T> &operator[](int i) {
    return m[i];
}
};

```

1.3.10 Modular Multiplicative Inverse

Time: $\mathcal{O}(\log m)$
Space: $\mathcal{O}(1)$

```

struct ModMultInv {

    // Fermat's Little Theorem: Used when m is prime
    int fermat(int a, int m) {
        BinaryExponentiation bin_exp;
        return bin_exp.run(a, m - 2);
    }

    // Extended Euclidean Algorithm: Used when m
    // and a are coprime
    int extended_euclidean(int a, int m) {
        ExtendedEuclidean ext_gcd;
        int x, y;
        int g = ext_gcd.run(a, m, x, y);
        return (x % m + m) % m;
    }
};

```

1.3.11 Sieve of Eratosthenes

Time: $\mathcal{O}(n \times \log \log n)$
Space: $\mathcal{O}(n)$

```

struct Sieve {
    int N;
    vector<int> is_prime;

    Sieve(int N) :
        N(N), is_prime(N+1)
    { init(); }

    void init() {
        fill(all(is_prime), 1);
    }

    vector<int> run() {
        vector<int> primes;
        init();

        for (int p = 2; p*p <= N; ++p)
            if (is_prime[p])
                for (int i = p*p; i <= N; i += p)
                    is_prime[i] = false;

        for (int p = 2; p <= N; ++p)
            if (is_prime[p])
                primes.pb(p);
    }
};

```

```

return primes;
}
};

```

1.4 Paradigm

1.4.1 Edit Distance

Time: $\mathcal{O}(m \times n)$
Space: $\mathcal{O}(m \times n)$

```

int dp[MAX][MAX];

struct EditDistance {
    int run(string a, string b) {
        for (int i = 0; i <= a.size(); ++i)
            for (int j = 0; j <= b.size(); ++j)
                if (i == 0)
                    dp[i][j] = j;
                else if (j == 0)
                    dp[i][j] = i;
                else if (a[i-1] == b[j-1])
                    dp[i][j] = d[i-1][j-1];
                else
                    dp[i][j] = 1 + min({dp[i][j-1],
                                         dp[i-1][j],
                                         dp[i-1][j-1]});

        return dp[a.size()][b.size()];
    }
};

```

1.4.2 Kadane

Time: $\mathcal{O}(n + m)$
Space: $\mathcal{O}(n + m)$

```

struct Kadane {
    int run(const vector<int> &v, int &start, int &end) {
        start = end = 0;
        int msf = -inf, meh = 0, s = 0;

        for (int i = 0; i < v.size(); ++i) {
            meh += v[i];

            if (msf < meh) {
                msf = meh;
                start = s, end = i;
            }

            if (meh < 0) {
                meh = 0;
                s = i + 1;
            }
        }

        return msf;
    }
};

```

1.4.3 Longest Increasing Subsequence (LIS)

Time: $\mathcal{O}(n^2)$

Space: $\mathcal{O}(n)$

```
struct LIS {
    int run(vector<int> v) {
        vector<int> lis(v.size()); lis[0] = 1;

        for (int i = 1; i < v.size(); ++i) {
            lis[i] = 1;

            for (int j = 0; j < i; ++j)
                if (v[i] > v[j] && lis[i] < lis[j] + 1)
                    lis[i] = lis[j] + 1;
        }

        return *max_element(all(lis));
    }
};
```

1.4.4 Longest Common Subsequence

Time: $\mathcal{O}(n \times m)$

Space: $\mathcal{O}(n \times m)$

```
int dp[MAX][MAX];

struct LCS {
    string run(string a, string b) {
        for (int i = 0; i <= a.size(); ++i) {
            for (int j = 0; j <= b.size(); ++j) {
                if (i == 0 || j == 0)
                    dp[i][j] = 0;
                else if (a[i - 1] == b[j - 1])
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                else
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }

        // The size is already at dp[n][m], now the common
        // subsequence is retrieved:

        int idx = dp[a.size()][b.size()];
        string ans(idx, ' ');

        int i = a.size(), j = b.size();
        while (i > 0 && j > 0) {
            if (a[i - 1] == b[j - 1]) {
                ans[idx - 1] = a[i - 1];
                i--, j--, idx--;
            } else if (dp[i - 1][j] > dp[i][j - 1])
                i--;
            else
                j--;
        }

        return ans;
    }
};
```

1.4.5 Ternary Search

Time: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(1)$

```
struct TernarySearch {
    // Unimodal function
    double f(double x) {
        return x * x;
    }

    double run(double l, double r) {
        double rt, lt;

        for (int i = 0; i < 500; ++i) {
            if (fabs(r - l) < EPS)
                return (l + r) / 2.0;

            lt = (r - l) / 3.0 + l;
            rt = ((r - l) * 2.0) / 3.0 + l;

            // '<' for minimum of f,
            // '>' for maximum of f
            if (f(lt) < f(rt))
                l = lt;
            else
                r = rt;
        }

        return (l + r) / 2.0;
    }
};
```

1.5 String

1.5.1 Knuth-Morris-Pratt (KMP)

Time:

- preprocess: $\mathcal{O}(m)$
- search: $\mathcal{O}(n)$

Space: $\mathcal{O}(n + m)$

```
struct KMP {
    string patt;
    vector<int> table;

    KMP(string patt) :
        patt(patt), table(patt.size()+1)
    { preprocess(); }

    void preprocess() {
        fill(all(table), -1);

        for (int i = 0, j = -1; i < patt.size(); ++i) {
            while (j >= 0 && patt[i] != patt[j])
                j = table[j];
            table[i + 1] = ++j;
        }
    }
};
```

```
}

vector<int> search(const string &txt) {
    vector<int> occurs;

    for (int i = 0, j = 0; i < txt.size(); ++i) {
        while (j >= 0 && txt[i] != patt[j])
            j = table[j];
        j++;

        if (j == patt.size()) {
            occurs.pb(i - j);
            j = table[j];
        }
    }

    return occurs;
}
};
```

1.5.2 Z-function

Time: $\mathcal{O}(n)$

Space: $\mathcal{O}(n)$

```
struct ZFunction {
    vector<int> run(string s) {
        int n = (int) s.length();
        vector<int> z(n);

        int l = 0, r = 0;
        for (int i = 1; i < n; ++i) {
            if (i <= r)
                z[i] = min(r - i + 1, z[i - l]);

            while (i + z[i] < n && s[z[i]] == s[i + z[i]])
                z[i]++;

            if (i + z[i] - 1 > r) {
                l = i;
                r = i + z[i] - 1;
            }
        }

        return z;
    }
};
```

1.6 Structure

1.6.1 AVL tree

Time: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(n)$

```
struct AVL {
    struct Node {
        int key, height;
        Node *left, *right;
    };
};
```

```

Node(int k) :
    key(k), height(1),
    left(nullptr), right(nullptr)
{}

static int get_height(Node *n) {
    return (n == nullptr) ? 0 : n->height;
}

void fix_state() {
    height = max(Node::get_height(left),
        Node::get_height(right)) + 1;
}

int get_balance() {
    return Node::get_height(left) -
        Node::get_height(right);
}
};

Node *root;

AVL() : root(nullptr) {}

void insert(int key) {
    root = insert(root, key);
}

private:

Node *rotate_right(Node *n) {
    Node *aux1 = n->left;
    Node *aux2 = aux1->right;
    aux1->right = n; aux1->fix_state();
    n->left = aux2; n->fix_state();
    return aux1;
}

Node *rotate_left(Node *n) {
    Node *aux1 = n->right;
    Node *aux2 = aux1->left;
    aux1->left = n; aux1->fix_state();
    n->right = aux2; n->fix_state();
    return aux1;
}

Node *insert(Node *n, int key) {
    if (n == nullptr) {
        Node *new_node = new Node(key);
        if (root == nullptr) root = new_node;
        return new_node;
    }

    if (key < n->key)
        n->left = insert(n->left, key);
    else
        n->right = insert(n->right, key);

    int balance = n->get_balance();
    n->fix_state();

    if (balance > 1 && key < n->left->key) {
        return rotate_right(n);
    } else if (balance < -1 && key > n->right->key) {

```

```

        return rotate_left(n);
    } else if (balance > 1 && key > n->left->key) {
        n->left = rotate_left(n->left);
        return rotate_right(n);
    } else if (balance < -1 && key < n->right->key) {
        n->right = rotate_right(n->right);
        return rotate_left(n);
    }

    return n;
}
};

```

1.6.2 Binary Indexed Tree (BIT)

Time:

- update: $\mathcal{O}(\log n)$
- query: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(n)$

```

struct BIT {
    int N;
    vector<int> tree;

    BIT(int N) :
        N(N), tree(N)
    { init(); }

    void init() { fill(all(tree), 0); }

    int query(int idx) {
        int sum = 0;
        for (; idx > 0; idx -= (idx & -idx))
            sum += tree[idx];
        return sum;
    }

    void update(int idx, int val) {
        for (; idx < N; idx += (idx & -idx))
            tree[idx] += val;
    }
};

```

1.6.3 Binary Indexed Tree 2D (BIT2D)

Time:

- update: $\mathcal{O}(\log^2 n)$
- query: $\mathcal{O}(\log^2 n)$

Space: $\mathcal{O}(n^2)$

```

struct BIT2D {
    int N, M;
    vector<vector<int>> tree;

    BIT2D(int N, int M) :
        N(N), M(M), tree(N, vector<int>(M))

```

```

{ init(); }

void init() {
    for (auto &i : tree)
        fill(all(i), 0);
}

int query(int idx, int idy) {
    int sum = 0;
    for (; idx > 0; idx -= (idx & -idx))
        for (int m = idy; m > 0; m -= (m & -m))
            sum += tree[idx][m];
    return sum;
}

void update(int idx, int idy, int val) {
    for (; idx < N; idx += (idx & -idx))
        for (int m = idy; m < M; m += (m & -m))
            tree[idx][m] += val;
}
};

```

1.6.4 Bitmask

Time: $\mathcal{O}(1)$

Space: $\mathcal{O}(1)$

```

struct Bitmask {
    ll state;

    Bitmask(ll state) :
        state(state) {}

    void set(int pos) {
        state |= (1 << pos);
    }

    void set_all(int n) {
        state = (1 << n) - 1;
    }

    void unset(int pos) {
        state &= ~(1 << pos);
    }

    void unset_all() {
        state = 0;
    }

    int get(int pos) {
        return state & (1 << pos);
    }

    void toggle(int pos) {
        state ^= (1 << pos);
    }

    int least_significant_one() {
        return state & (-state);
    }
};

```

1.6.5 Disjoint-set

Time:

- `make_set`: $\mathcal{O}(1)$
- `find_set`: $\mathcal{O}(a(n))$
- `union_set`: $\mathcal{O}(a(n))$

Space: $\mathcal{O}(n)$

```
struct DisjointSet {
    int N;
    vector<int> rank, par;

    DisjointSet(int N) :
        N(N), rank(N), par(N)
    { init(); }

    void init() {
        iota(all(par), 0);
        fill(all(rank), 0);
    }

    int find_set(int x) {
        if (par[x] != x)
            par[x] = find_set(par[x]);
        return par[x];
    }

    void union_set(int x, int y) {
        x = find_set(x);
        y = find_set(y);

        if (x != y) {
            if (rank[x] > rank[y]) swap(x, y);
            if (rank[x] == rank[y]) rank[x]++;
            par[x] = y;
        }
    }
};
```

1.6.6 Lazy Segment Tree

Time:

- `build_tree`: $\mathcal{O}(n \log n)$
- `update_tree`: $\mathcal{O}(\log n)$
- `query_tree`: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(n)$

```
int N;
struct LazySegmentTree {
    vector<int> tree, lazy;

    LazySegmentTree(const vector<int> &v) :
        tree(MAX*4), lazy(MAX*4)
    {
        init();
        build(v);
    }
};
```

```
void init() {
    fill(all(tree), 0);
    fill(all(lazy), 0);
}

inline int left(int x) { return (x << 1); }
inline int right(int x) { return (x << 1) + 1; }

void build(const vector<int> &v, int node = 1,
           int a = 0, int b = N - 1)
{
    if (a > b)
        return;

    if (a == b) {
        tree[node] = v[a];
        return;
    }

    int mid = (a + b) / 2;
    build(v, left(node), a, mid);
    build(v, right(node), mid + 1, b);
    tree[node] = tree[left(node)] + tree[right(node)];
}

void push(int node, int a, int b, int val) {
    tree[node] += val;
    // tree[node] += (b - a + 1)*val; (for Range Sum Query)

    if (a != b) {
        lazy[left(node)] += val;
        lazy[right(node)] += val;
    }

    lazy[node] = 0;
}

void update(int i, int j, int val, int node = 1,
            int a = 0, int b = N - 1)
{
    if (lazy[node] != 0)
        push(node, a, b, lazy[node]);

    if (a > b || a > j || b < i)
        return;

    if (i <= a && b <= j) {
        push(node, a, b, val);
        return;
    }

    int mid = (a + b) / 2;
    update(i, j, val, left(node), a, mid);
    update(i, j, val, right(node), mid + 1, b);
    tree[node] = tree[left(node)] + tree[right(node)];
}

int query(int i, int j, int node = 1,
          int a = 0, int b = N - 1)
{
    if (a > b || a > j || b < i)
        return 0;

    if (lazy[node])
```

```
    push(node, a, b, lazy[node]);

    if (a >= i && b <= j)
        return tree[node];

    int mid = (a + b) / 2;
    int q1 = query(i, j, left(node), a, mid);
    int q2 = query(i, j, right(node), mid + 1, b);
    return q1 + q2;
}
};
```

1.6.7 Policy Tree

Description:

A set-like STL structure with order statistics.

Time:

- `insert`: $\mathcal{O}(\log n)$
- `erase`: $\mathcal{O}(\log n)$
- `find_by_order`: $\mathcal{O}(\log n)$
- `order_of_key`: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(n)$

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

using namespace __gnu_pbds;

typedef tree<
    int,
    null_type,
    less<int>,
    rb_tree_tag,
    tree_order_statistics_node_update
> set_t;

void operations() {
    set_t S;

    S.insert(x);
    S.erase(x);

    // Return iterator to the k-th largest element
    // (counting from zero)
    int pos = *S.find_by_order(k);

    // Return the number of items strictly smaller
    // than x
    int ord = S.order_of_key(x)
}
```

1.6.8 Segment Tree

Time:

- `update`: $\mathcal{O}(\log n)$
- `query`: $\mathcal{O}(\log n)$

Space: $\mathcal{O}(n)$

```
struct SegmentTree {
    int N;
    vector<int> tree;

    Tree(int N) :
        N(N), tree(2 * N, 0) {}

    // Base depends on 'op':
    // op: a + b -> base: 0
    // op: min(a,b) -> base: inf
    int base = -inf;
    int op(int a, int b) {
        return max(a, b);
    }

    void update(int idx, int val) {
        idx += N;
        tree[idx] = val;

        while (idx > 1) {
            tree[idx / 2] = op(tree[idx & ~1], tree[idx | 1]);
            idx /= 2;
        }
    }

    int query(int l, int r) {
        int ra = base, rb = base;
        l += N, r += N;

        while (l < r) {
            if (l % 2) ra = op(ra, tree[l++]);
            if (r % 2) rb = op(tree[--r], rb);

            l >>= 1;
            r >>= 1;
        }

        return op(ra, rb);
    }
};
```

1.6.9 Sqrt Decomposition

Time:

- preprocess: $\mathcal{O}(n)$
- query: $\mathcal{O}(\sqrt{n})$
- update: $\mathcal{O}(1)$

Space: $\mathcal{O}(n)$

```
struct SqrtDecomposition {
    int block_size;
    vector<int> v, block;

    SqrtDecomposition(vector<int> v) :
        v(v), block(v.size())
    { init(); }

    void init() {
        preprocess(v.size());
    }

    void update(int idx, int val) {
        block[idx / block_size] += val - v[idx];
        v[idx] = val;
    }

    int query(int l, int r) {
        int ans = 0;

        for (; l < r && ((l % block_size) != 0); ++l)
            ans += v[l];

        for (; l + block_size <= r; l += block_size)
            ans += block[l / block_size];

        for (; l <= r; ++l)
            ans += v[l];

        return ans;
    }

    void preprocess(int n) {
        block_size = sqrt(n);

        int idx = -1;
        for (int i = 0; i < n; ++i) {
            if (i % block_size == 0)
                block[++idx] = 0;

            block[idx] += v[i];
        }
    }
};
```

1.6.10 Trie

Time:

- insert: $\mathcal{O}(M)$
- search: $\mathcal{O}(M)$

Space: $\mathcal{O}(\text{alph} \times N)$

```
template <typename T>
struct Trie {
    int states;

    vector<int> ending;
    vector<vector<int>> trie;

    // Number of words (N) and number of letters per word
    // (M), and number of letters in alphabet (alph).
    Trie(int N, int M, int alph) :
        ending(N * M),
        trie(N * M, vector<int>(alph))
    { init(); }

    void init() {
        states = 0;
        for (auto &i : trie)
            fill(all(i), -1);
    }

    int len(T x) {
        if constexpr(is_same_v<T,int>)
            return 32;
        return x.size();
    }

    int idx(T x) {
        if constexpr(is_same_v<T,int>)
            return !(x & (1 << i));
        return x[i] - 'a';
    }

    void insert(T x) {
        int node = 0;

        for (int i = 0; i < len(x); ++i) {
            if (trie[node][idx(x, i)] == -1)
                trie[node][idx(x, i)] = ++states;
            node = trie[node][idx(x, i)];
        }

        ending[node] = true;
    }

    bool search(T x) {
        int node = 0;

        for (int i = 0; i < len(x); ++i) {
            node = trie[node][idx(x, i)];
            if (node == -1)
                return false;
        }

        return ending[node];
    }
};
```

2 Misc

2.1 Environment

2.1.1 Vim Config

```
" Tabs
set expandtab
set smarttab

" Indents
set shiftwidth=2
set tabstop=2
set autoindent
set smartindent
set cindent

" Turn backup off
set nobackup
set nowb
```

```
set noswapfile

" Highlight matching brackets
set showmatch

" Display line numbers
set number
```

2.1.2 Template

```
#define EPS 1e-6
#define MOD 1000000007
#define inf 0x3f3f3f3f
#define llinf 0x3f3f3f3f3f3f3f3f
#define fi first
```

```
#define se second
#define pb push_back
#define ende '\n'

#define all(x) (x).begin(), (x).end()
#define rall(x) (x).rbegin(), (x).rend()
#define mset(x, y) memset(&x, (y), sizeof(x))

using namespace std;

using ll = long long;
using ii = pair<int,int>;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    return 0;
}
```