

Contents

1	Algorithms	2
1.1	Geometry	2
1.1.1	Circle	2
1.1.2	Convex Hull	2
1.1.3	Point	2
1.1.4	Polygon	2
1.1.5	Geometry Primitives	3
1.1.6	Segment	3
1.2	Graph	3
1.2.1	Articulations and Bridges	3
1.2.2	Bellman-Ford	3
1.2.3	Bipartite Matching	3
1.2.4	Centroid Decomposition	4
1.2.5	Dijkstra	4
1.2.6	Dinic's	4
1.2.7	Edmonds-Karp	5
1.2.8	Floyd Warshall	5
1.2.9	Ford-Fulkerson	5
1.2.10	Heavy Light Decomposition	5
1.2.11	Hopcroft-Karp	6
1.2.12	Kosaraju	6
1.2.13	Kruskal	7
1.2.14	Lowest Common Ancestor (LCA)	7
1.2.15	Minimum Cost Maximum Flow	7
1.2.16	Prim	8
1.2.17	Steiner Tree	8
1.2.18	Tarjan	8
1.2.19	Topological Sort	9
1.2.20	Travelling Salesman	9
1.3	Math	9
1.3.1	Big Integer	9
1.3.2	Binary Exponentiation	11
1.3.3	Chinese Remainder Theorem	11
1.3.4	Euler Totient (ϕ)	11
1.3.5	Extended Euclidean algorithm	12
1.3.6	Fast Fourier Transform (FFT)	12
1.3.7	Gale-Shapley (Stable Marriage)	12
1.3.8	Karatsuba	13
1.3.9	Legendre's Formula	13
1.3.10	Linear Diophantine Equation	13

CONTENTS		2
1.3.11	Matrix	13
1.3.12	Miller-Rabin primality test	14
1.3.13	Modular Multiplicative Inverse	14
1.3.14	Pollard's Rho	14
1.3.15	Sieve of Eratosthenes	14
1.4	Paradigm	14
1.4.1	Dynamic Programming, Divide and Conquer Optimization	14
1.4.2	Edit Distance	14
1.4.3	Kadane	15
1.4.4	Longest Increasing Subsequence (LIS)	15
1.4.5	Longest Common Subsequence	15
1.4.6	Ternary Search	15
1.5	String	15
1.5.1	Aho-Corasick	15
1.5.2	Booth's Algorithm	16
1.5.3	Knuth-Morris-Pratt (KMP)	16
1.5.4	Z-function	16
1.6	Structure	16
1.6.1	AVL tree	16
1.6.2	Binary Indexed Tree (BIT)	17
1.6.3	Binary Indexed Tree 2D (BIT2D)	17
1.6.4	Bitmask	17
1.6.5	Disjoint-set	17
1.6.6	Hash Function (splitmix64)	18
1.6.7	Lazy Segment Tree	18
1.6.8	Mo's Algorithm	18
1.6.9	Policy Tree	19
1.6.10	Segment Tree	19
1.6.11	2D Segment Tree	19
1.6.12	Sparse Table	20
1.6.13	Sqrt Decomposition	20
1.6.14	Trie	20
2	Misc	21
2.1	Environment	21
2.1.1	Vim Config	21
2.1.2	Template	21

1 Algorithms

1.1 Geometry

1.1.1 Circle

```
struct Circle {
    Point<> c;
    double r;

    Circle(Point<> c, double r) : c(c), r(r) {}

    // Circumcircle
    Circle(Point<> a, Point<> b, Point<> c) {
        Point<> u((b - a).y, -(b - a).x);
        Point<> v((c - a).y, -(c - a).x);
        Point<> n = (c - b)*0.5;

        double t = u.cross(n) / v.cross(u);

        this->c = (a + c)*0.5 + v*t;
        this->r = dist(this->c, a);
    }

    // Minimum enclosing circle: O(n)
    Circle(vector<Point<>> p) {
        random_shuffle(all(p));
        Circle C(p[0], 0.0);

        for (int i = 0; i < p.size(); ++i) {
            if (C.contains(p[i])) continue;
            C = Circle(p[i], 0.0);

            for (int j = 0; j < i; ++j) {
                if (C.contains(p[j])) continue;
                C = Circle((p[j] + p[i])*0.5, 0.5*dist(p[j], p[i]));

                for (int k = 0; k < j; ++k) {
                    if (C.contains(p[k])) continue;
                    C = Circle(p[j], p[i], p[k]);
                }
            }
        }

        this->c = C.c;
        this->r = C.r;
    }

    bool contains(Point<> p) {
        return (dist(c, p) <= r + EPS);
    }
};
```

1.1.2 Convex Hull

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

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

```
template <typename T>
bool cw(Point<T> a, Point<T> b, Point<T> c) {
```

```
    return (b - a).cross(c - a) <= 0;
}

template <typename T>
vector<Point<T>> convex_hull(vector<Point<T>> &v) {
    int k = 0;
    vector<Point<T>> ans(v.size() * 2);

    sort(all(v), [](const Point<T> &a, const Point<T> &b) {
        return (a.x == b.x) ? (a.y < b.y) : (a.x < b.x);
    });

    for (int i = 0; i < v.size(); ++i) {
        for (; k >= 2 && cw(ans[k-2], ans[k-1], v[i]); --k);
        ans[k++] = v[i];
    }

    for (int i = v.size() - 2, t = k + 1; i >= 0; --i) {
        for (; k >= 2 && cw(ans[k-2], ans[k-1], v[i]); --k);
        ans[k++] = v[i];
    }

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

1.1.3 Point

```
template <typename T = double>
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); }
    Point operator*(T s) { return Point(x*s, y*s); }

    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;
    }
};
```

1.1.4 Polygon

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

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

    // Adds a vertex to the polygon.
    void add_point(Point<T> 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;
    }

    // Rotating Calipers
    double width() {
        vector<Point<T>> h = convex_hull(v);

        int n = h.size() - 1;
        double ans = 1e14;

        h[0] = h[n];
        for (int i = 1, j = 1; i <= n; ++i) {
            while ((h[i] - h[i-1]).cross(h[j%n+1] - h[i-1]) >
                    (h[i] - h[i-1]).cross(h[j] - h[i-1]))
                j = j % n + 1;

            Segment<T> seg(h[i], h[i-1]);
            ans = min(ans, seg.dist(h[j]));
        }

        return ans;
    }

    // Rotating Calipers
    double diameter() {
        vector<Point<T>> h = convex_hull(v);

        if (h.size() == 1) return 0;
        if (h.size() == 2) return dist(h[0], h[1]);

        int n = h.size() - 1;
```

```

double ans = -1e14;

h[0] = h[n];
for (int i = 1, j = 1; i <= n; ++i) {
    while ((h[i] - h[i-1]).cross(h[j%n+1] - h[i-1]) >
           (h[i] - h[i-1]).cross(h[j] - h[i-1]))
        j = j % n + 1;

    ans = max(ans, dist(h[j], h[i]));
    ans = max(ans, dist(h[j], h[i-1]));
}

return ans;
};

```

1.1.5 Geometry Primitives

```

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

template <typename T>
double dist(Point<T> a, Point<T> b) {
    return hypot(a.x - b.x, a.y - b.y);
}

```

1.1.6 Segment

```

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

    Segment(Point<T> a, Point<T> b) : a(a), b(b) {}

    // Checks if points p and q are on the same side
    // of the segment.
    bool same_side(Point<T> p, Point<T> 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<T> 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));
    }

    // Distance between segment and point
    double dist(Point<T> p) {
        return (a - b).cross(p - b) / sqrt((b - a).dot(b - a));
    }

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

```

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<p<int>> brid;
    vector<int> arti;

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

    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(p<int>(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: $\mathcal{O}(V \cdot E)$
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)
    { init(); }

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

    void add_edge(int u, int v, int w) {
        graph.pb({ u, v, w });
    }

    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] + e.w < dist[e.v])
                return -inf;

        return dist[d];
    }
};

```

1.2.3 Bipartite Matching

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

```

vector<int> graph[MAX];

struct BipartiteMatching {

```

```

int N;
vector<int> vis, match;

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

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 $O(n \log n)$ paths (from each node to all of its ancestors in the centroid decomposition).

Time: $O(V \log V)$

Space: $O(V + E)$

```

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

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

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

    void init() {

```

```

        fill(all(vis), 0);
        build(0); // 0-indexed vertices
    }

    void build(int x, int p = -1) {
        int n = dfs(x);
        int c = get_centroid(x, n);
        vis[c] = 1;
        par[c] = p;

        for (auto i : graph[c])
            if (!vis[i])
                build(i, c);
    }

    // Calculates size of every subtree.
    int dfs(int x, int p = -1) {
        size[x] = 1;
        for (auto i : graph[x])
            if (i != p && !vis[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 && !vis[i])
                return get_centroid(i, n, x);
        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: $O(E + V \log V)$

Space: $O(V + E)$

```

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<p<int>> pq;
        dist[s] = 0;
        pq.insert({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({dist[i.fi], i.fi});
                }
            }

            return dist[d];
        }
    };
};

```

1.2.6 Dinic's

Time: $O(E \cdot V^2)$

Space: $O(V + E)$

```

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 u, int v, int c) {
        Edge forw = { v, 0, c, (int) graph[v].size() };
        Edge back = { u, 0, 0, (int) graph[u].size() };
        graph[u].pb(forw);
        graph[v].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: $\mathcal{O}(V \cdot E^2)$

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];

void floyd_warshall(int n) {
    memset(dist, inf);

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (graph[i][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: $\mathcal{O}(E \cdot f)$

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 Heavy Light Decomposition

Description:

The Heavy Light Decomposition splits a tree into several path so that we can reach the root from any node by traversing at most $\log n$ paths. For every vertice v , an edge is called heavy if it leads to a child with the largest subtree size; all other edges are called light.

The set of disjoint paths containing heavy edges are called the “heavy chains”. These paths (both heavy and light) can be indexed by a single segment tree, allowing fast queries and updates to edges (or vertices). And also, information about the chains (such as the head), allows for fast traversals from any node to any of its ancestors.

Time:

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

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

Caution:

- Modifications are necessary if values are associated with vertices;

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

```
template <typename ST>
struct HLD {
    ST &seg;
    int cnum, ptr;

    // depth, parent, value of node
    vector<int> dep, par, val;

    // head[i]: head of i-th node's chain;
    // heavy[i]: "special child" of i-th node
    // pos[i]: position of i-th node on segtree
    // bot[i]: bottommost (depth-wise) node on the i-th
    // edge (required only when edges have weights)
    vector<int> head, heavy, pos, bot;
```

```
HLD(int n, ST &seg) :
    seg(seg),
    dep(n, 0), par(n), val(n),
    head(n), heavy(n, -1), pos(n), bot(n)
{
    cnum = ptr = 0;
    N = n; // global N for segtree
    dfs(0);
    decompose(0);

    // (required only when edges have weights)
    for (int i = 0; i < n - 1; ++i)
        if (dep[edge[i].fi] > dep[edge[i].se])
            bot[i] = edge[i].fi;
        else
            bot[i] = edge[i].se;
}
```

```
int dfs(int x, int p = -1) {
    int size = 1;
    par[x] = p;

    int max_size = 0;
    for (auto i : graph[x])
        if (i.fi != p) {
            dep[i.fi] = dep[x] + 1;
            val[i.fi] = i.se;
            int isize = dfs(i.fi, x);

            size += isize;
            if (isize > max_size)
                max_size = isize, heavy[x] = i.fi;
        }

    return size;
}
```

```
void decompose(int x, int h = 0) {
    head[x] = h;
    seg.update(ptr, val[x]);
    pos[x] = ptr++;
}
```

```
    if (heavy[x] != -1)
        decompose(heavy[x], h);

    for (auto i : graph[x])
        if (i.fi != par[x] && i.fi != heavy[x])
            decompose(i.fi, i.fi);
}

// Queries max edge (or vertex) on the path
// between a and b
int query(int a, int b) {
    int ans = seg.ident;
    for (; head[a] != head[b]; b = par[head[b]]) {
        if (dep[head[a]] > dep[head[b]])
            swap(a, b);
        ans = seg.func(ans, seg.query(pos[head[b]], pos[b]));
    }

    if (dep[a] > dep[b]) swap(a, b);

    // Remove "+ 1" when values are associated with
    // vertices
    return seg.func(ans, seg.query(pos[a] + 1, pos[b]));
}

// Updates value of i-th edge (or vertex)
void update(int i, int val) {
    seg.update(pos[bot[i]], val);
}
};
```

1.2.11 Hopcroft-Karp

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

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

Caution:

- Assumes 0-indexed vertices in graph;

```
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), -1);
        fill(all(matchR), -1);
    }

    bool bfs() {
        queue<int> Q;
        for (int l = 0; l < L; ++l)
            if (matchL[l] == -1) {
                dist[l] = 0;
```

```
                Q.push(l);
            } else
                dist[l] = -1;

    bool ans = false;
    while (!Q.empty()) {
        int l = Q.front(); Q.pop();
        for (auto r : graph[l])
            if (matchR[r] == -1)
                ans = true;
            else if (dist[matchR[r]] == -1) {
                dist[matchR[r]] = dist[l] + 1;
                Q.push(matchR[r]);
            }
    }

    return ans;
}

bool dfs(int l) {
    if (l == -1)
        return true;

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

    return false;
}

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

    return ans;
}
};
```

1.2.12 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;
    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;
}
};

```

1.2.13 Kruskal

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

```

using edge = pair<int, int>;
vector<edge> edges;

struct Kruskal {
    int N;
    DisjointSet ds;

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

    // Minimum Spanning Tree: comp = less<int>()
    // Maximum Spanning Tree: comp = greater<int>()
    int run(vector<edge> &mst, function<bool(int, int)> comp) {

```

```

        sort(all(edges), [&](const edge &a, const edge &b) {
            return comp(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 (pu != pv) {
                mst.pb(edges[i]);
                size += edges[i].se;
                ds.union_set(pu, pv);
            }
        }

        return size;
    }
};

```

1.2.14 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 LOG 20
#define COST 0

vector<p<int>> graph[MAX];
int par[MAX][LOG], cost[MAX][LOG];

template <typename T, class F = function<T(T,T)>>
struct LCA {
    F func;
    vector<int> h;

    LCA(int N, F func) : h(N), func(func) {
        init();
    }

    void init() {
        mset(par, -1);
        mset(cost, 0);
        dfs(0);
    }
}

```

```

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 < LOG; ++i)
        if (par[v][i-1] != -1) {
            par[v][i] = par[par[v][i-1]][i-1];
            cost[v][i] = func(cost[v][i-1], func(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 a, int b) {
    int ans = 0;
    if (h[a] < h[b]) swap(a, b);

    for (int i = LOG - 1; i >= 0; --i)
        if (par[a][i] != -1 && h[par[a][i]] >= h[b]) {
            ans = func(ans, cost[a][i]);
            a = par[a][i];
        }

    if (a == b) {
#ifdef COST
        return ans;
    #else
        return a;
    #endif
    }

    for (int i = LOG - 1; i >= 0; --i)
        if (par[a][i] != -1 && par[a][i] != par[b][i]) {
            ans = func(ans, func(cost[a][i], cost[b][i]));
            a = par[a][i];
            b = par[b][i];
        }

#ifdef COST
    if (a == b)
        return ans;
    else
        return func(ans, func(cost[a][0], cost[b][0]));
#else
    return par[a][0];
#endif
};

```

1.2.15 Minimum Cost Maximum Flow

Time: $\mathcal{O}(V^2 \cdot E)$
Space: $\mathcal{O}(V + E)$

```

struct MinCostMaxFlow {
    struct Edge { int u, v, cap, cost; };

    vector<Edge> edges;

```



```

vector<vector<int>>> adj;
vector<int> vis, dist, par, ind;

MinCostMaxFlow(int N) :
    vis(N), dist(N), par(N), ind(N), adj(N) {}

void add_edge(int u, int v, int cap, int cost) {
    adj[u].pb(edges.size());
    edges.pb({ u, v, cap, cost });
    adj[v].pb(edges.size());
    edges.pb({ v, u, 0, -cost });
}

// Shortest Path Faster Algorithm (slower than
// Dijkstra but works with negative edges).
bool spfa(int s, int t) {
    fill(all(dist), inf);
    dist[s] = 0;
    queue<int> Q;
    Q.push(s);

    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        vis[u] = 0;

        for (auto i : adj[u]) {
            Edge &e = edges[i];
            int v = e.v;
            if (e.cap > 0 && dist[v] > dist[u] + e.cost) {
                dist[v] = dist[u] + e.cost;
                par[v] = u;
                ind[v] = i;

                if (!vis[v]) {
                    Q.push(v);
                    vis[v] = 1;
                }
            }
        }
    }

    return dist[t] < inf;
}

// Returns pair (min_cost, max_flow).
p<int> run(int s, int t) {
    int min_cost = 0;
    int max_flow = 0;

    while (spfa(s, t)) {
        int flow = inf;
        for (int i = t; i != s; i = par[i])
            flow = min(flow, edges[ind[i]].cap);

        for (int i = t; i != s; i = par[i]) {
            edges[ind[i]].cap -= flow;
            edges[ind[i]^1].cap += flow;
        }

        min_cost += flow * dist[t];
        max_flow += flow;
    }

    return {min_cost, max_flow};
}
};

```

1.2.16 Prim

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

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

```

vector<p<int>> 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<p<int>> pq;
        for (auto i : graph[0])
            pq.push({-i.se, -i.fi});

        int ans = 0;
        while (!pq.empty()) {
            p<int> 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({-i.se, -i.fi});
            }
        }

        return ans;
    }
};

```

1.2.17 Steiner Tree

Description:

A (minimum) Steiner tree is a tree that, given a graph G and a set of terminal vertices T (inside the graph), connects all vertices in T using other vertices in G if necessary, and minimizes the sum of weights of the included edges.

The algorithm uses dynamic programming, where $dp[i][mask]$ stores the value of a Steiner tree rooted at i containing the terminal nodes represented by the bits in $mask$. The algorithm iterates over all bitmasks, and, at each iteration $mask$, every pair of complementary subsets are tested and $dp[i][mask]$ is updated with the value given by the combination of both trees. Then, still at step $mask$,

$dp[j][mask]$ is updated for every j with the “extension” of i (for every i), as if the root was moving around.

The result must be retrieved from a node (root of the final Steiner tree) that contains all terminal nodes and has the smallest value.

Time: $\mathcal{O}(n^2 \cdot 2^t + n \cdot 3^t)$

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

```

int dist[MAXN][MAXN];
int graph[MAXN][MAXN];
int dp[MAXN][1 << MAXT];

int steiner_tree(int n, int t) {
    floyd_warshall(n);

    mset(dp, inf);
    for (int i = 0; i < t; ++i)
        dp[i][1 << i] = 0;

    for (int mask = 1; mask < (1 << t); ++mask) {
        for (int i = 0; i < n; ++i)
            for (int ss = mask; ss > 0; ss = (ss - 1) & mask)
                dp[i][ss] = min(dp[i][mask],
                                dp[i][ss] + dp[i][mask ^ ss]);

        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                dp[j][mask] = min(dp[j][mask],
                                    dp[i][mask] + dist[i][j]);
    }

    int ans = inf;
    for (int i = 0; i < n; ++i)
        ans = min(ans, dp[i][(1 << t) - 1]);

    return ans;
}

```

1.2.18 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;
}
};

```

1.2.19 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;
}
};

```

1.2.20 Travelling Salesman

Description:

Given a graph and an origin vertex, this algorithm returns the shortest possible route that visits each vertex and returns to the origin.

The algorithm works by using dynamic programming, where $dp[i][mask]$ stores the last visited vertex i and a set of visited vertices represented by a bitmask $mask$. Given a state, the next vertex in the path is chosen by a recursive call, until the bitmask is full, in which case the weight of the edge between the last vertex and the origin is returned.

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

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

```

int dp[MAX][1 << MAX];
int graph[MAX][MAX];

struct TSP {
    int N;

    TSP(int N) : N(N)
    { init(); }

    void init() { memset(dp, -1); }

    int solve(int i, int mask) {
        if (mask == (1 << N) - 1)
            return graph[i][0];
        if (dp[i][mask] != -1)
            return dp[i][mask];

        int ans = inf;
        for (int j = 0; j < N; ++j)
            if (!(mask & (1 << j)) && (i != j))
                ans = min(ans, graph[i][j] +
                    solve(j, mask | (1 << j)));
    }
}

```

```

return dp[i][mask] = ans;
}

int run(int start) {
    return solve(start, 1 << start);
}
};

```

1.3 Math

1.3.1 Big Integer

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

Caution:

- Just use Python if possible.

```

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

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

    BigInt() : sign(1) {}
    BigInt(i64 x) { *this = x; }
    BigInt(const string &s) { read(s); }

    void operator=(const BigInt &x) {
        sign = x.sign;
        num = x.num;
    }

    void operator=(i64 x) {
        sign = 1;
        if (x < 0) sign = -1, x = -x;
        for (; x > 0; x /= base)
            pb(x % base);
    }

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

        int carry = 0;
        BigInt res = x;

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

                res[i] += carry + (i < size() ? num[i] : 0);
                carry = res[i] >= base;
                if (carry) res[i] -= base;
            }

        return res;
    }

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

```

```

    int carry = 0;
    BigInt res = *this;

    for (int i = 0; i < x.size() || carry; ++i) {
        res[i] -= carry + (i < x.size() ? x[i] : 0);
        carry = res[i] < 0;
        if (carry) res[i] += base;
    }

    res.trim();
    return res;
}

void operator*=(int x) {
    if (x < 0) sign = -sign, x = -x;

    int carry = 0;
    for (int i = 0; i < size() || carry; ++i) {
        if (i == size()) pb(0);
        i64 cur = num[i] * (i64) x + carry;

        carry = (int) (cur / base);
        num[i] = (int) (cur % base);
    }

    trim();
}

BigInt operator*(int x) const {
    BigInt res = *this;
    res *= x;
    return res;
}

friend pair<BigInt, BigInt> divmod(const BigInt &a1,
    const BigInt &b1)
{
    int norm = base / (b1.back() + 1);
    BigInt a = a1.abs() * norm;
    BigInt b = b1.abs() * norm;
    BigInt q, r;
    q.resize(a.size());

    for (int i = a.size() - 1; i >= 0; i--) {
        r *= base;
        r += a[i];

        int s1 = r.size() <= b.size() ? 0 : r[b.size()];
        int s2 = r.size() <= b.size() - 1 ? 0 : r[b.size() - 1];
        int d = ((i64) base * s1 + s2) / b.back();

        r -= b * d;
        while (r < 0) r += b, --d;
        q[i] = d;
    }

    q.sign = a1.sign * b1.sign;
    r.sign = a1.sign;
    q.trim(); r.trim();

    return make_pair(q, r / norm);
}

```

```

BigInt operator/(const BigInt &x) const {
    return divmod(*this, x).fi;
}

BigInt operator%(const BigInt &x) const {
    return divmod(*this, x).se;
}

void operator/=(int x) {
    if (x < 0) sign = -sign, x = -x;

    for (int i = size() - 1; rem = 0; i >= 0; --i) {
        i64 cur = num[i] + rem * (i64) base;
        num[i] = (int) (cur / x);
        rem = (int) (cur % x);
    }

    trim();
}

BigInt operator/(int x) const {
    BigInt res = *this;
    res /= x;
    return res;
}

int operator%(int x) const {
    if (x < 0) x = -x;

    int m = 0;
    for (int i = size() - 1; i >= 0; --i)
        m = (num[i] + m * (i64) base) % x;

    return m * sign;
}

void operator+=(const BigInt &x) { *this = *this + x; }
void operator-=(const BigInt &x) { *this = *this - x; }
void operator*=(const BigInt &x) { *this = *this * x; }
void operator/=(const BigInt &x) { *this = *this / x; }

bool operator<(const BigInt &x) const {
    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] * sign;

    return false;
}

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) && !(x < *this);
}

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

void trim() {
    while (!empty() && !back()) pop_back();
    if (empty()) sign = 1;
}

bool is_zero() const {
    return empty() || (size() == 1 && !num[0]);
}

BigInt operator-() const {
    BigInt res = *this;
    res.sign = -sign;
    return res;
}

BigInt abs() const {
    BigInt res = *this;
    res.sign *= res.sign;
    return res;
}

i64 to_long() const {
    i64 res = 0;
    for (int i = size() - 1; i >= 0; i--)
        res = res * base + num[i];
    return res * sign;
}

friend BigInt gcd(const BigInt &a, const BigInt &b) {
    return b.is_zero() ? a : gcd(b, a % b);
}

friend BigInt lcm(const BigInt &a, const BigInt &b) {
    return a / gcd(a, b) * b;
}

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

    int pos = 0;
    while (pos < 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;
}

static vector<int> convert_base(
    const vector<int> &a,
    int oldd, int newd) {
    vector<i64> p(max(oldd, newd) + 1);
    p[0] = 1;
    for (int i = 1; i < p.size(); i++)
        p[i] = p[i - 1] * 10;

    i64 cur = 0;
    int curd = 0;
    vector<int> res;

    for (int i = 0; i < a.size(); i++) {
        cur += a[i] * p[curd];
        curd += oldd;

        while (curd >= newd) {
            res.pb(int(cur % p[newd]));
            cur /= p[newd];
            curd -= newd;
        }
    }

    res.pb((int) cur);
    while (!res.empty() && !res.back())
        res.pop_back();
    return res;
}

BigInt operator*(const BigInt &x) const {
    vector<int> a6 = convert_base(this->num, base_d, 6);
    vector<int> b6 = convert_base(x.num, base_d, 6);

    vector<i64> a(all(a6));
    vector<i64> b(all(b6));

    while (a.size() < b.size()) a.pb(0);
    while (b.size() < a.size()) b.pb(0);
    while (a.size() & (a.size() - 1))
        a.pb(0), b.pb(0);

    vector<i64> c = karatsuba(a, b);

    BigInt res;
    int carry = 0;
    res.sign = sign * x.sign;

    for (int i = 0; i < c.size(); i++) {

```

```

        i64 cur = c[i] + carry;
        res.pb((int) (cur % 1000000));
        carry = (int) (cur / 1000000);
    }

    res.num = convert_base(res.num, 6, base_d);
    res.trim();
    return res;
}

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

void pop_back() { num.pop_back(); }
void resize(int x) { num.resize(x); }
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

Description:

Computes fast exponentiation by looking at the binary representation of the exponent. The usage of “bin_mul” is not necessary, but it ensures that no overflow will occur when multiplying two large integers. It is definitely required in order to work with Miller-Rabin and Pollard’s Rho implementations (when dealing with numbers that might go up to 10^{18} . USE BIN_MUL ONLY IF NECESSARY (VERY SLOW).

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

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

```

i64 bin_mul(i64 a, i64 b, i64 M = MOD) {
    i64 x = 0;
    a %= M;
    while (b) {
        if (b & 1) x = (x + a) % M;
        b >>= 1;
        a = (a * 2) % M;
    }
    return x % M;
}

i64 bin_exp(i64 a, i64 e, i64 M = MOD) {
    i64 x = 1;
    while (e) {
        if (e & 1)
            x = (x * a) % M;
        e >>= 1;
        a = (a * a) % M;
    }
    return x % M;
}

```

1.3.3 Chinese Remainder Theorem

Description:

Given t linear congruences in the format:

$$x \equiv a_i \pmod{m_i}$$

the Chinese Remainder Theorem (CRT) finds x that satisfies every given congruence or states that there are no solutions.

This implementation does not require all m_i to be coprime, it works with any value and returns the smallest possible x ; infinitely many solutions can be obtained by incrementing or decrementing $LCM(m_1, m_2, \dots, m_t)$ from x .

Time: $\mathcal{O}(t \log LCM(m_1, m_2, \dots, m_t))$

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

Caution:

- It is very easy to get overflow, since the LCM is computed from all m_i , BigInt or Python should be considered if inputs are too large

```

i64 norm(i64 a, i64 b) {
    a %= b;
    return (a < 0) ? a + b : a;
}

pair<i64, i64> crt_single(i64 a, i64 n, i64 b, i64 m) {
    ans_t e = ext_gcd(n, m);

    if ((a - b) % e.d != 0)
        return {-1, -1}; // No solution

    i64 lcm = (m/e.d) * n;
    i64 ans = norm(a + e.x*(b-a) / e.d % (m/e.d)*n, lcm);
    return {norm(ans, lcm), lcm};
}

i64 crt(vector<i64> a, vector<i64> m) {
    i64 ans = a[0];
    i64 lcm = m[0];

    int t = a.size();
    for (int i = 1; i < t; ++i) {
        auto ss = crt_single(ans, lcm, a[i], m[i]);
        if (ss.fi == -1)
            return -1; // No solution

        ans = ss.fi;
        lcm = ss.se;
    }

    return ans;
}

```

1.3.4 Euler Totient (ϕ)

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

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

```
i64 phi(i64 n) {
    i64 ans = n;

    for (i64 i = 2; i*i <= n; i++)
        if (n % i == 0) {
            while (n % i == 0) n /= i;
            ans -= ans / i;
        }

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

    return ans;
}
```

1.3.5 Extended Euclidean algorithm

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

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

```
struct ans_t { i64 x, y, d; };

ans_t ext_gcd(i64 a, i64 b) {
    if (a == 0) return {0, 1, b};
    ans_t e = ext_gcd(b % a, a);
    return {e.y - (b/a)*e.x, e.x, e.d};
}
```

1.3.6 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<Complex> 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);

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

        // Build 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;

        fa[i] = z;
    }

    fft(fa);

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

    return res;
}

};
```

1.3.7 Gale-Shapley (Stable Marriage)

Description:

Two groups, each of size N are given: men and women. Each person has a list of preference ranking all N people of the opposite sex. The task is to unite both groups into stable pairs. A set of pairs is stable if there are no unassigned couple that like each other more than their assigned pair.

The algorithm's steps are: 1) allow every men to propose to their highest ranking woman; 2) the women become tentatively engaged to their top choice of men; 3) all rejected men propose to their next choice; 4) the woman replaces their current pair in case a man with a higher rank proposes to her, the replaced men are now marked as rejected; 5) repeat step 3 until all men are paired.

The result is guaranteed to return a configuration where every man gets the best possible wife, while every woman gets the worst possible husband (it benefits the group that chooses first).

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

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

Caution:

- Men are indexed by $[0, N)$
- The result prioritizes men, swapping the bottom half of the matrix by the top half will invert who's given preference

```
// Receives matrix of preferences pref[2*N][N] and returns
// vector v where v[m] contains preference of the m-th man.
vector<int> gale_shapley(const vector<vector<int>> &pref) {
    int n = pref[0].size();
    vector<int> w_part(n, -1);
    vector<int> m_part(n, -1);
    vector<int> start(n, 0);
```

```

while (true) {
    int m;
    for (m = 0; m < n; ++m)
        if (m_part[m] == -1)
            break;

    if (m == n) break;

    for (; start[m] < n && m_part[m] == -1; ++start[m]) {
        int w = pref[m][start[m]];

        if (w_part[w - n] == -1) {
            w_part[w - n] = m;
            m_part[m] = w;
        } else {
            int m1 = w_part[w - n];
            bool pref_m = false;

            for (int j = 0; j < n; ++j)
                if (pref[w][j] == m) {
                    pref_m = true;
                    break;
                } else if (pref[w][j] == m1)
                    break;

            if (pref_m) {
                w_part[w - n] = m;
                m_part[m] = w;
                m_part[m1] = -1;
            }
        }
    }

    return m_part;
}

```

1.3.8 Karatsuba

Time: $\mathcal{O}(n^{\log(3)})$

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

```

vector<i64> karatsuba(const vector<i64> &a,
                    const vector<i64> &b)
{
    int n = a.size();
    vector<i64> res(n + n);

    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                res[i + j] += a[i] * b[j];

        return res;
    }

    int k = n >> 1;
    vector<i64> a1(a.begin(), a.begin() + k);
    vector<i64> a2(a.begin() + k, a.end());
    vector<i64> b1(b.begin(), b.begin() + k);
    vector<i64> b2(b.begin() + k, b.end());

    vector<i64> a1b1 = karatsuba(a1, b1);

```

```

vector<i64> a2b2 = karatsuba(a2, b2);

for (int i = 0; i < k; i++)
    a2[i] += a1[i];
for (int i = 0; i < k; i++)
    b2[i] += b1[i];

vector<i64> r = karatsuba(a2, b2);
for (int i = 0; i < a1b1.size(); i++)
    r[i] -= a1b1[i];
for (int i = 0; i < a2b2.size(); i++)
    r[i] -= a2b2[i];

for (int i = 0; i < r.size(); i++)
    res[i + k] += r[i];
for (int i = 0; i < a1b1.size(); i++)
    res[i] += a1b1[i];
for (int i = 0; i < a2b2.size(); i++)
    res[i + n] += a2b2[i];

return res;
}

```

1.3.9 Legendre's Formula

Description:

Given an integer n and a prime number p , find the largest power of p (x) such that $n!$ is divisible by p^x .

Writing $n!$ as $1 \cdot 2 \cdot 3 \dots (n-1) \cdot n$ shows that every k -th element of the product is divisible by k (i.e. adds 1 to the answer), the number of such elements is $\lfloor \frac{n}{k} \rfloor$. The same is true for every k^i , thus, the answer is given by:

$$\lfloor \frac{n}{k} \rfloor + \lfloor \frac{n}{k^2} \rfloor + \dots + \lfloor \frac{n}{k^i} \rfloor$$

only approximately the first $\log_p n$ elements are greater than zero, therefore the sum is finite.

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

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

```

int legendre(int n, int p) {
    int x = 0;

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

    return x;
}

```

1.3.10 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() {
        ans_t e = ext_gcd(a, b); d = e.d;
        if (c % d == 0) {
            x0 = e.x * (c / d);
            y0 = e.y * (c / d);
            has_solution = true;
        } else
            has_solution = false;
    }

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

```

1.3.11 Matrix

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

```

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];
    }
};
```

1.3.12 Miller-Rabin primality test

Time: $\mathcal{O}(k \cdot \log^3 n)$
Space: $\mathcal{O}(1)$

```
const vector<i64> A = {
    2, 325, 9375, 28178, 450775, 9780504, 1795265022
};

bool is_prime(i64 n) {
    if (n < 2 || n % 6 % 4 != 1)
        return n - 2 < 2;

    i64 s = __builtin_ctzll(n - 1);
    i64 d = n >> s;

    for (auto a : A) {
        i64 p = bin_exp(a, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--)
            p = bin_mul(p, p, n);
        if (p != n - 1 && i != s)
            return 0;
    }

    return 1;
}
```

1.3.13 Modular Multiplicative Inverse

Description:
Given integers a and m , the modular multiplicative inverse of a is an integer x such that

$$a \cdot x \equiv 1 \pmod{m}$$

also denoted as a^{-1} .

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

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

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

1.3.14 Pollard’s Rho

Description:
Returns prime factorization of n .

Time: $\mathcal{O}(n^{1/4})$
Space: $\mathcal{O}(1)$

```
i64 pollard(i64 n) {
    auto f = [n](i64 x) {
        return (bin_mul(x, x, n) + 1) % n;
    };

    if (n % 2 == 0) return 2;

    for (i64 i = 2; ; ++i) {
        i64 x = i, y = f(x), p;
        while ((p = __gcd(n + y - x, n)) == 1)
            x = f(x), y = f(f(y));
        if (p != n) return p;
    }
}

vector<i64> factor(i64 n) {
    if (n == 1) return {};
    if (is_prime(n)) // Use Miller-Rabin
        return {n};

    i64 x = pollard(n);
    auto l = factor(x);
    auto r = factor(n/x);

    l.insert(l.end(), all(r));
    return l;
}
```

1.3.15 Sieve of Eratosthenes

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

```
vector<int> sieve(int n) {
    vector<int> primes;
    vector<int> is_prime(n + 1, 1);

    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 Dynamic Programming, Divide and Conquer Optimization

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

```
vector<int> v;
int dp[102][20101];

void solve(int g, int i, int j, int l, int r) {
    int m = (i + j) / 2;
    if (i > j) return;

    p<int> best = { -inf, -1 };
    for (int k = l; k <= min(r, m); ++k) {
        mx = max(mx, v[k]);
        mn = min(mn, v[k]);
        best = max(best, { dp[g-1][k-1] + cost(k, m, k) });
    }

    dp[g][m] = best.fi;
    solve(g, i, m - 1, l, best.se);
    solve(g, m + 1, j, best.se, r);
}

int calc() {
    // setup dp[0][i];

    for (int i = 1; i <= k; ++i)
        solve(i, 1, n, 1, n);

    return dp[k][n];
}
```

1.4.2 Edit Distance

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

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

int edit_distance(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()];
}
```

1.4.3 Kadane

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

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

```
int kadane(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;
}
```

1.4.4 Longest Increasing Subsequence (LIS)

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

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

```
int lis(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));
}
```

1.4.5 Longest Common Subsequence

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

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

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

string lcs(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;
}
```

1.4.6 Ternary Search

Description:

Searches for the minimum (or maximum) value of a unimodal function $f(x)$ for x between l and r .

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

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

Caution:

- f must be a unimodal function

```
// Minimum of f: comp = less<T>()
// Maximum of f: comp = greater<T>()
template <typename T, class F = function<T(T)>>
T ternary_search(T l, T r, F f, function<bool(T,T)> comp) {
    T rt, lt;

    for (int i = 0; i < 500; ++i) {
        if (fabs(r - l) < EPS)
            return (l + r) / 2;

        lt = (r - l) / 3 + l;
        rt = ((r - l) * 2) / 3 + l;

        if (comp(f(lt), f(rt)))
            l = lt;
        else
            r = rt;
    }

    return (l + r) / 2;
}
```

1.5 String

1.5.1 Aho-Corasick

Time:

- build: $\mathcal{O}(n)$
- preprocess: $\mathcal{O}(n)$
- match: $\mathcal{O}(p + n)$
- match_all: $\mathcal{O}(p + x)$

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

Caution:

- Match might not find all occurrences when repeated strings are given (fix with map)

```
struct AhoCorasick {
    struct Node {
        vector<int> words;
        map<char, int> next;
        int fail, cnt, hei, occ;

        Node() : fail(0), cnt(0), hei(0), occ(-1) {}
        int has(char i) { return next.count(i); }
        int &operator[](char i) { return next[i]; }
    };

    vector<int> top;
    vector<Node> trie;

    AhoCorasick(const vector<string> &v) {
        trie.pb(Node());
        build(v);
        top = preprocess();
    }

    int insert(const string &s) {
        int n = 0;
        for (int i = 0; i < s.size(); n = trie[n][s[i]], ++i)
            if (!trie[n].has(s[i])) {
                trie[n][s[i]] = trie.size();
                trie[n].hei = i + 1;
                trie.pb(Node());
            }
        return n;
    }

    void build(const vector<string> &v) {
        for (int i = 0; i < v.size(); ++i) {
            int n = insert(v[i]);
            trie[n].words.pb(i);
        }
    }

    inline int suffix(int v, char c) {
        while (v != 0 && !trie[v].has(c)) v = trie[v].fail;
        if (trie[v].has(c)) v = trie[v][c];
        return v;
    }

    vector<int> preprocess() {
```



```

vector<int> Q = { 0 };
for (int i = 0; i != Q.size(); ++i) {
    int u = Q[i];
    for (auto j : trie[u].next) {
        int &v = trie[j.se].fail;
        if (u) {
            v = suffix(trie[u].fail, j.fi);
            trie[j.se].occ = trie[v].words.size() ? v :
                trie[v].occ;
        } else {
            v = trie[u].fail;
            trie[j.se].occ = -1;
        }
        Q.pb(j.se);
    }
}
return Q;

// Returns vector with indices of the strings occuring at
// least once in pattern p
vector<int> match(const string &p) {
    int u = 0;
    for (auto i : p) {
        u = suffix(u, i);
        trie[u].cnt++;
    }
    for (int i = top.size() - 1; i >= 0; --i)
        trie[trie[top[i]].fail].cnt += trie[top[i]].cnt;

    vector<int> ans;
    for (auto &i : trie)
        if (i.cnt && i.words.size())
            for (auto j : i.words) ans.pb(j);

    sort(all(ans));
    return ans;
}

// Returns all occurrences of strings in p, where ans[i].fi
// is the indice of the string and ans[i].se is where in p
// does the string start
vector<p<int>> match_all(const string &p) {
    int u = 0;
    vector<p<int>> ans;
    for (int i = 0; i < p.size(); ++i) {
        u = suffix(u, p[i]);
        for (auto j : trie[u].words)
            ans.pb({j, i - trie[u].hei + 1});

        int x = u;
        while (trie[x].occ != -1) {
            x = trie[x].occ;
            for (auto j : trie[x].words)
                ans.pb({j, i - trie[x].hei + 1});
        }
    }
    sort(all(ans));
    return ans;
}
};

```

1.5.2 Booth's Algorithm

Description:

This algorithm finds the lexicographically minimal (or maximal) string rotation, that is, given a string s , the goal is to find a rotation of s possessing the lowest (or highest) lexicographical order among all possible rotations.

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

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

```

// Maximal: func = greater<char>()
// Minimal: func = less<char>()
string booth(string s, function<bool(char, char)> func) {
    string S = s + s;
    vector<int> f(S.size(), -1);

    int k = 0;
    for (int j = 1; j < S.size(); ++j) {
        char sj = S[j];
        int i = f[j - k - 1];

        while (i != -1 && sj != S[k+i+1]) {
            if (func(sj, S[k+i+1]))
                k = j - i - 1;
            i = f[i];
        }

        if (sj != S[k+i+1]) {
            if (func(sj, S[k+i+1]))
                k = j;

            f[j-k] = -1;
        } else
            f[j-k] = i + 1;
    }

    string ans(s.size(), 0);
    for (int i = 0; i < s.size(); ++i)
        ans[i] = S[k+i];
    return ans;
}

```

1.5.3 Knuth-Morris-Pratt (KMP)

Time:

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

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

```

struct KMP {
    string patt;
    vector<int> pi;

    KMP(string patt) :
        patt(patt), pi(patt.size())
    { preprocess(); }

    void preprocess() {

```

```

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

            if (patt[i] == patt[j]) j++;
            pi[i] = j;
        }

        void search(const string &txt) {
            for (int i = 0, j = 0; i < txt.size(); ++i) {
                while (j > 0 && txt[i] != patt[j])
                    j = pi[j - 1];

                if (txt[i] == patt[j]) j++;
                if (j == patt.size()) {
                    cout << "Pattern found at " << (i - j) << endl;
                    j = pi[j - 1];
                }
            }
        }
};

```

1.5.4 Z-function

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

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

```

vector<int> z_function(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, 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, 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 {
    i64 state;

    Bitmask(i64 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> rnk, par, siz;

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

    void init() {
        iota(all(par), 0);
        fill(all(rnk), 0);
        fill(all(siz), 1);
    }

    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) return;
        if (rnk[x] < rnk[y]) swap(x, y);
        if (rnk[x] == rnk[y]) rnk[x]++;
        par[y] = x;
        siz[x] += siz[y];
    }
};
```

1.6.6 Hash Function (splitmix64)

Description:

The default hash function for `unordered_map` in C++ is not reliable when it comes to non-random input (i.e. input designed to cause collisions, such as multiples of a specific prime). This custom hash function solves this problem (<https://codeforces.com/blog/entry/62393>).

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

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

```
struct CustomHash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::
            steady_clock::now()
            .time_since_epoch()
            .count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

```
};

unordered_map<i64,int,CustomHash> M;
```

1.6.7 Lazy Segment Tree

Time:

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

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

Caution:

- Provide value for N before any operation

```
#define left(x) (x << 1)
#define right(x) ((x << 1) + 1)

int N;

template <typename T, class F = function<T(const T&, const T&)>>
struct LazySegmentTree {
    F func;
    T id = T();
    vector<T> tree, lazy;

    LazySegmentTree(F func) :
        func(func), tree(MAX*4, 0), lazy(MAX*4, 0) {}

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

        if (l == r)
            tree[node] = v[l];
        else {
            int m = (l + r) / 2;
            build(v, left(node), l, m);
            build(v, right(node), m + 1, r);
            tree[node] = func(tree[left(node)], tree[right(node)]);
        }

        void push(int node, int l, int r, T val) {
            tree[node] += (r - l + 1) * val;

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

            lazy[node] = 0;
        }

        void update(int i, int j, T val,
                   int node = 1, int l = 0, int r = N - 1)
    {
```

```
        if (lazy[node] != 0)
            push(node, l, r, lazy[node]);

        if (l > r || l > j || r < i) return;

        if (i <= l && r <= j)
            push(node, l, r, val);
        else {
            int m = (l + r) / 2;
            update(i, j, val, left(node), l, m);
            update(i, j, val, right(node), m + 1, r);
            tree[node] = func(tree[left(node)], tree[right(node)]);
        }

        T query(int i, int j,
               int node = 1, int l = 0, int r = N - 1)
    {
        if (l > r || l > j || r < i) return id;

        if (lazy[node])
            push(node, l, r, lazy[node]);

        if (l >= i && r <= j)
            return tree[node];

        int m = (l + r) / 2;
        T q1 = query(i, j, left(node), l, m);
        T q2 = query(i, j, right(node), m + 1, r);
        return func(q1, q2);
    }
};
```

1.6.8 Mo's Algorithm

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

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

Caution:

- Remember to implement add, remove, and get_ans functions.

```
struct Query {
    int l, r, idx;
};

vector<int> mos_algorithm(vector<Query> Q) {
    int blk_size = (int) sqrt(v.size() + 0.0) + 1;

    vector<int> ans(Q.size());
    sort(all(Q), [](Query a, Query b) {
        return p<int>(a.l / blk_size, a.r) < p<int>(b.l /
            blk_size, b.r);
    });

    int curr_l = 0, curr_r = -1;

    for (auto q : Q) {
        while (curr_l > q.l) {
            curr_l--;
            add(curr_l);
        }
        while (curr_r < q.r) {
```

```

        curr_r++;
        add(curr_r);
    }
    while (curr_l < q.l) {
        remove(curr_l);
        curr_l++;
    }
    while (curr_r > q.r) {
        remove(curr_r);
        curr_r--;
    }

    ans[q.idx] = get_ans();
}

return ans;
}

```

1.6.9 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;

template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void operations() {
    ordered_set S;

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

    // Return iordered_set terator 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)
}

```

1.6.10 Segment Tree

Time:

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

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

Caution:

- Provide identity value if necessary (default is T())

```

template<typename T, class F = function<T(T,T)>>
struct SegmentTree {
    int N;
    F func;
    T id = T();
    vector<T> tree;

    SegmentTree(int N, F func) :
        N(N), func(func), tree(N*2, id)
    {}

    void build(const vector<T> &v) {
        for (int i = 0; i < N; ++i)
            tree[N + i] = v[i];
        for (int i = N - 1; i > 0; --i)
            tree[i] = func(tree[i*2], tree[i*2+1]);
    }

    void update(int i, T x) {
        tree[i += N] = x;
        for (i /= 2; i > 0; i /= 2)
            tree[i] = func(tree[i*2], tree[i*2+1]);
    }

    T query(int i, int j) {
        T left = id, right = id;
        for (i += N, j += N; i < j; i /= 2, j /= 2) {
            if (i & 1) left = func(left, tree[i++]);
            if (j & 1) right = func(right, tree[--j]);
        }
        return func(left, right);
    }
};

```

1.6.11 2D Segment Tree

Time:

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

Space: $\mathcal{O}(16 \cdot n \cdot m)$

Caution:

- Very high constant in space complexity

```

#define left(x) (x << 1)
#define right(x) ((x << 1) + 1)

int N, M;

template<typename T, class F = function<T(T,T)>>
struct SegmentTree2D {
    using matrix<T> = vector<vector<T>>;

```

```

    F func;
    T id = T();
    matrix<T> tree;

    SegmentTree2D(F func) :
        tree(4*MAX, vector<T>(4*MAX, 0)),
        func(func) {}

    void build_row(const matrix<T> &mat,
        int ni, int li, int ri,
        int nj = 1, int lj = 0, int rj = M - 1)
    {
        if (lj == rj) {
            if (li == ri)
                tree[ni][nj] = mat[li][lj];
            else
                tree[ni][nj] = func(tree[left(ni)][nj], tree[
                    right(ni)][nj]);
        } else {
            int m = (lj + rj) / 2;
            build_row(mat, ni, li, ri, left(nj), lj, m);
            build_row(mat, ni, li, ri, right(nj), m+1, rj);
            tree[ni][nj] = func(tree[ni][left(nj)], tree[ni][
                right(nj)]);
        }
    }

    void build(const matrix<T> &mat,
        int ni = 1, int li = 0, int ri = N - 1)
    {
        if (li != ri) {
            int m = (li + ri) / 2;
            build(mat, left(ni), li, m);
            build(mat, right(ni), m+1, ri);
        }
        build_row(mat, ni, li, ri);
    }

    T query_row(int j1, int j2, int i,
        int nj = 1, int lj = 0, int rj = M - 1)
    {
        if (lj > rj || lj > j2 || rj < j1) return id;

        if (j1 <= lj && rj <= j2)
            return tree[i][nj];

        int m = (lj + rj) / 2;
        T q1 = query_row(j1, j2, i, left(nj), lj, m);
        T q2 = query_row(j1, j2, i, right(nj), m + 1, rj);
        return func(q1, q2);
    }

    T query(int i1, int j1, int i2, int j2,
        int ni = 1, int li = 0, int ri = N - 1)
    {
        if (li > ri || li > i2 || ri < i1) return id;

        if (i1 <= li && ri <= i2)
            return query_row(j1, j2, ni);

        int m = (li + ri) / 2;
        T q1 = query(i1, j1, i2, j2, left(ni), li, m);
        T q2 = query(i1, j1, i2, j2, right(ni), m + 1, ri);
        return func(q1, q2);
    }
}

```

```

}

void update_row(int i, int j, T val,
               int ni, int li, int ri,
               int nj = 1, int lj = 0, int rj = M - 1)
{
    if (lj > rj || lj > j || rj < j) return;

    if (lj == rj) {
        if (li == ri)
            tree[ni][nj] = val;
        else
            tree[ni][nj] = func(tree[left(ni)][nj], tree[
                right(ni)][nj]);
    } else {
        int m = (lj + rj) / 2;
        update_row(i, j, val, ni, li, ri, left(nj), lj, m);
        update_row(i, j, val, ni, li, ri, right(nj), m+1,
            rj);
        tree[ni][nj] = func(tree[ni][left(nj)], tree[ni][
            right(nj)]);
    }
}

void update(int i, int j, T val,
           int ni = 1, int li = 0, int ri = N - 1)
{
    if (li > ri || li > i || ri < i) return;

    if (li != ri) {
        int m = (li + ri) / 2;
        update(i, j, val, left(ni), li, m);
        update(i, j, val, right(ni), m+1, ri);
    }

    update_row(i, j, val, ni, li, ri);
}
};

```

1.6.12 Sparse Table

Time:

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

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

Caution:

- func must be min, max, gcd, or lcm

```
#define LOG 20
```

```

template <typename T, class F = function<T(T,T)>>
struct SparseTable {
    F func;
    int N;
    vector<T> log;
    vector<vector<T>> table;

    SparseTable(const vector<T> &v, F func) :
        N(v.size()),

```

```

    log(N + 1),
    table(N, vector<T>(LOG+1)),
    func(func)
{ preprocess(v); }

void preprocess(const vector<T> &v) {
    log[1] = 0;
    for (int i = 2; i <= N; ++i)
        log[i] = log[i >> 1] + 1;

    for (int i = 0; i < N; ++i)
        table[i][0] = v[i];

    for (int j = 1; j <= LOG; ++j)
        for (int i = 0; i + (1 << j) <= N; ++i)
            table[i][j] = func(table[i][j-1], table[i+(1<<
                j-1)][j-1]);
}

T query(int l, int r) {
    int j = log[r - l + 1];
    return func(table[l][j], table[r - (1<<j) + 1][j]);
}
};

```

1.6.13 Sqrt Decomposition

Time:

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

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

```

struct SqrtDecomposition {
    int blk_size;
    vector<int> v, blk;

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

    void init() {
        build();
    }

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

    int query(int l, int r) {
        int ans = 0;
        int cl = l/blk_size, cr = r/blk_size;

        if (cl == cr) {
            for (int i = l; i <= r; ++i)
                ans += v[i];
        } else {
            for (int i = l, end=(cl+1)*blk_size-1; i <= end; ++
                i)

```

```

        ans += v[i];
        for (int i = cl+1; i <= cr - 1; ++i)
            ans += blk[i];
        for (int i = cr*blk_size; i <= r; ++i)
            ans += v[i];
    }

    return ans;
}

void build() {
    int n = v.size();
    blk_size = (int) sqrt(n + 0.0) + 1;
    for (int i = 0; i < n; ++i)
        blk[idx / blk_size] += v[i];
}
};

```

1.6.14 Trie

Time:

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

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

Caution:

- If Trie stores integers, remember to check order of bits (most/least significant first).

```

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];
}
};

```

```

#ifdef LOCAL
#include <local.h>
#endif

#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 i64 = long long;
template <typename T> using p = pair<T,T>;

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

    return 0;
}

```

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

```
#include <bits/stdc++.h>
```