



# Notebook - Competitive Programming

Anões do TLE

## Contents

<b>1 Data structures</b>	<b>2</b>	4.8 Floyd Warshall . . . . .	8	5.14 Sieve . . . . .	13
1.1 Matrix . . . . .	2	4.9 Graph . . . . .	8	5.15 Sum Divisors . . . . .	13
1.2 Merge Sort Tree . . . . .	3	4.10 TopSort - Kahn . . . . .	8	5.16 Sum of difference . . . . .	14
1.3 Minimal Excluded With Updates (MEX-U) . . . . .	4	4.11 Kruskal . . . . .	9	<b>6 Problems</b>	<b>14</b>
1.4 Minimal Excluded (MEX) . . . . .	4	4.12 Minimax . . . . .	9	6.1 Kth Digit String (CSES) . . . . .	14
1.5 Segment Tree (Parameterized OP) . . . . .	4	4.13 MSF . . . . .	9	<b>7 Strings</b>	<b>14</b>
1.6 Segment Tree 2D . . . . .	4	4.14 Minimum Spanning Graph (MSG) . . . . .	9	7.1 Edit Distance . . . . .	14
1.7 Union Find Disjoint Set (UFDS) . . . . .	5	4.15 Prim . . . . .	10	7.2 Manacher . . . . .	14
<b>2 Dynamic programming</b>	<b>5</b>	4.16 Retrieve Path 2d . . . . .	10	7.3 Rabin Karp . . . . .	15
2.1 Kadane . . . . .	5	4.17 Retrieve Path . . . . .	10	<b>8 Trees</b>	<b>15</b>
2.2 Longest Increasing Subsequence (LIS) . . . . .	5	4.18 Second Best MST . . . . .	10	8.1 LCA Binary Lifting (CP Algo) . . . . .	15
<b>3 Geometry</b>	<b>5</b>	4.19 TopSort - Tarjan . . . . .	11	8.2 LCA SegTree (CP Algo) . . . . .	15
3.1 Convex Hull . . . . .	5	<b>5 Math</b>	<b>11</b>	8.3 LCA Sparse Table . . . . .	16
3.2 Point To Segment . . . . .	6	5.1 Binomial . . . . .	11	8.4 Tree Flatten . . . . .	16
<b>4 Graphs</b>	<b>6</b>	5.2 Count Divisors Range . . . . .	11	8.5 Tree Isomorph . . . . .	17
4.1 Articulation Points . . . . .	6	5.3 Count Divisors . . . . .	11	<b>9 Settings and macros</b>	<b>17</b>
4.2 Bellman Ford . . . . .	7	5.4 Factorization With Sieve . . . . .	11	9.1 short-macro.cpp . . . . .	17
4.3 BFS 0/1 . . . . .	7	5.5 Factorization . . . . .	12	9.2 macro.cpp . . . . .	17
4.4 Bridges . . . . .	7	5.6 Fast Doubling - Fibonacci . . . . .	12	<b>10 Theoretical guide</b>	<b>18</b>
4.5 Negative Cycle Bellman Ford . . . . .	7	5.7 Fast Exp Iterative . . . . .	12	10.1 Modular Multiplicative Inverse . . . . .	18
4.6 Negative Cycle Floyd Warshall . . . . .	7	5.8 Fast Exp . . . . .	12	10.2 Exponent With Module . . . . .	18
4.7 Dijkstra . . . . .	8	5.9 GCD . . . . .	12	10.3 Notable Series . . . . .	18
		5.10 Integer Mod . . . . .	12		
		5.11 Is prime . . . . .	13		
		5.12 LCM . . . . .	13		
		5.13 Euler phi $\varphi(n)$ . . . . .	13		

# 1 Data structures

## 1.1 Matrix

```
template <typename T>
struct Matrix {
    vector<vector<T>> d;

    Matrix() : Matrix(0) {}
    Matrix(int n) : Matrix(n, n) {}
    Matrix(int n, int m) : Matrix(vector<vector<T>>(n, vector<T>(m))) {}
    Matrix(const vector<vector<T>> &v) : d(v) {}

    constexpr int n() const { return (int)d.size(); }
    constexpr int m() const { return n() ? (int)d[0].size() : 0; }

    void rotate() { *this = rotated(); }

    Matrix<T> rotated() const {
        Matrix<T> res(m(), n());
        for (int i = 0; i < m(); i++) {
            for (int j = 0; j < n(); j++) {
                res[i][j] = d[n() - j - 1][i];
            }
        }
        return res;
    }

    Matrix<T> pow(int power) const {
        assert(n() == m());

        auto res = Matrix<T>::identity(n());
        auto b = *this;
        while (power) {
            if (power & 1) res *= b;
            b *= b;
            power >>= 1;
        }
        return res;
    }

    Matrix<T> submatrix(int start_i, int start_j, int rows = INT_MAX,
                       int cols = INT_MAX) const {
        rows = min(rows, n() - start_i);
        cols = min(cols, m() - start_j);
        if (rows <= 0 or cols <= 0) return {};

        Matrix<T> res(rows, cols);
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++) res[i][j] = d[i + start_i][j + start_j];
        return res;
    }

    Matrix<T> translated(int x, int y) const {
        Matrix<T> res(n(), m());
        for (int i = 0; i < n(); i++) {
            for (int j = 0; j < m(); j++) {
                if (i + x < 0 or i + x >= n() or j + y < 0 or j + y >= m()) continue;

```

```
                res[i + x][j + y] = d[i][j];
            }
        }
        return res;
    }

    static Matrix<T> identity(int n) {
        Matrix<T> res(n);
        for (int i = 0; i < n; i++) res[i][i] = 1;
        return res;
    }

    vector<T> &operator[](int i) { return d[i]; }
    const vector<T> &operator[](int i) const { return d[i]; }
    Matrix<T> &operator+=(T value) {
        for (auto &row : d) {
            for (auto &x : row) x += value;
        }
        return *this;
    }
    Matrix<T> operator+(T value) const {
        auto res = *this;
        for (auto &row : res) {
            for (auto &x : row) x = x + value;
        }
        return res;
    }
    Matrix<T> &operator-=(T value) {
        for (auto &row : d) {
            for (auto &x : row) x -= value;
        }
        return *this;
    }
    Matrix<T> operator-(T value) const {
        auto res = *this;
        for (auto &row : res) {
            for (auto &x : row) x = x - value;
        }
        return res;
    }
    Matrix<T> &operator*=(T value) {
        for (auto &row : d) {
            for (auto &x : row) x *= value;
        }
        return *this;
    }
    Matrix<T> operator*(T value) const {
        auto res = *this;
        for (auto &row : res) {
            for (auto &x : row) x = x * value;
        }
        return res;
    }
    Matrix<T> &operator/=(T value) {
        for (auto &row : d) {
            for (auto &x : row) x /= value;
        }
        return *this;
    }
}
```

```

}
Matrix<T> operator/(T value) const {
    auto res = *this;
    for (auto &row : res) {
        for (auto &x : row) x = x / value;
    }
    return res;
}
Matrix<T> &operator+=(const Matrix<T> &o) {
    assert(n() == o.n() and m() == o.m());
    for (int i = 0; i < n(); i++) {
        for (int j = 0; j < m(); j++) {
            d[i][j] += o[i][j];
        }
    }
    return *this;
}
Matrix<T> operator+(const Matrix<T> &o) const {
    assert(n() == o.n() and m() == o.m());
    auto res = *this;
    for (int i = 0; i < n(); i++) {
        for (int j = 0; j < m(); j++) {
            res[i][j] = res[i][j] + o[i][j];
        }
    }
    return res;
}
Matrix<T> &operator-=(const Matrix<T> &o) {
    assert(n() == o.n() and m() == o.m());
    for (int i = 0; i < n(); i++) {
        for (int j = 0; j < m(); j++) {
            d[i][j] -= o[i][j];
        }
    }
    return *this;
}
Matrix<T> operator-(const Matrix<T> &o) const {
    assert(n() == o.n() and m() == o.m());
    auto res = *this;
    for (int i = 0; i < n(); i++) {
        for (int j = 0; j < m(); j++) {
            res[i][j] = res[i][j] - o[i][j];
        }
    }
    return res;
}
Matrix<T> &operator*=(const Matrix<T> &o) {
    *this = *this * o;
    return *this;
}
Matrix<T> operator*(const Matrix<T> &o) const {
    assert(m() == o.n());
    Matrix<T> res(n(), o.m());
    for (int i = 0; i < res.n(); i++) {
        for (int j = 0; j < res.m(); j++) {
            auto &x = res[i][j];
            for (int k = 0; k < m(); k++) {
                x += (d[i][k] * o[k][j]);
            }
        }
    }
}

```

```

    }
}
return res;
}

friend istream &operator>>(istream &is, Matrix<T> &mat) {
    for (auto &row : mat)
        for (auto &x : row) is >> x;
    return is;
}

friend ostream &operator<<(ostream &os, const Matrix<T> &mat) {
    bool frow = 1;
    for (auto &row : mat) {
        if (not frow) os << '\n';
        bool first = 1;
        for (auto &x : row) {
            if (not first) os << ' ';
            os << x;
            first = 0;
        }

        frow = 0;
    }
    return os;
}

auto begin() { return d.begin(); }
auto end() { return d.end(); }
auto rbegin() { return d.rbegin(); }
auto rend() { return d.rend(); }

auto begin() const { return d.begin(); }
auto end() const { return d.end(); }
auto rbegin() const { return d.rbegin(); }
auto rend() const { return d.rend(); }
};

```

## 1.2 Merge Sort Tree

Like a segment tree but each node  $st_i$  stores a sorted subarray

- $inrange(l, r, a, b)$  : counts the number of elements  $x \in [l, r]$  such that  $a \leq x \leq b$ .

Memory:  $O(N \log N)$

Build:  $O(N \log N)$

$inrange$ :  $O(\log^2 N)$

```

template <class T>
struct MergeSortTree {
    int n;
    vector<vector<T>> st;
    MergeSortTree(vector<T>& xs) : n(len(xs)), st(n << 1) {
        for (int i = 0; i < n; i++) st[i + n] = vector<T>({xs[i]});

        for (int i = n - 1; i > 0; i--) {
            st[i].resize(len(st[i << 1]) + len(st[i << 1 | 1]));
            merge(all(st[i << 1]), all(st[i << 1 | 1]), st[i].begin());
        }
    }
}

```

```

int count(int i, T a, T b) {
    return upper_bound(all(st[i]), b) - lower_bound(all(st[i]), a);
}

int inrange(int l, int r, T a, T b) {
    int ans = 0;

    for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
        if (l & 1) ans += count(l++, a, b);
        if (r & 1) ans += count(--r, a, b);
    }

    return ans;
}
};

```

### 1.3 Minimal Excluded With Updates (MEX-U)

In the problem you need to change individual numbers in the array, and compute the new MEX of the array after each such update.

Pre-compute:  $O(N \log N)$

Update:  $O(\log N)$

Query:  $O(1)$

```

class Mex {
private:
    map<ll, ll> frequency;
    set<ll> missing_numbers;
    vl A;

public:
    Mex(vl const& A) : A(A) {
        for (ll i = 0; i <= A.size(); i++) missing_numbers.insert(i);

        for (ll x : A) {
            ++frequency[x];
            missing_numbers.erase(x);
        }
    }

    ll mex() { return *missing_numbers.begin(); }

    void update(ll idx, ll new_value) {
        if (--frequency[A[idx]] == 0) missing_numbers.insert(A[idx]);
        A[idx] = new_value;
        ++frequency[new_value];
        missing_numbers.erase(new_value);
    }
};

```

### 1.4 Minimal Excluded (MEX)

Given an array  $A$  of size  $N$ . You have to find the minimal non-negative element that is not present in the array. That number is commonly called the MEX (minimal excluded).

Time:  $O(N)$

```

ll mex(vl const& A) {
    static bool used[MAX + 111] = {0};

    for (ll x : A) {
        if (x <= MAX) used[x] = true;
    }

    ll result = 0;
    while (used[result]) ++result;

    for (ll x : A) {
        if (x <= MAX) used[x] = false;
    }

    return result;
}

```

### 1.5 Segment Tree (Parameterized OP)

Query:  $O(\log N)$

Update:  $O(\log N)$

```

template <typename T, auto op>
class SegTree {
private:
    T e;
    ll N;
    vector<T> seg;

public:
    SegTree(ll N, T e) : e(e), N(N), seg(N + N, e) {}

    void assign(ll i, T v) {
        i += N;
        seg[i] = v;
        for (i >>= 1; i; i >>= 1) seg[i] = op(seg[2 * i], seg[2 * i + 1]);
    }

    T query(ll l, ll r) {
        T la = e, ra = e;
        l += N;
        r += N;

        while (l <= r) {
            if (l & 1) la = op(la, seg[l++]);
            if (~r & 1) ra = op(seg[r--], ra);
            l >>= 1;
            r >>= 1;
        }

        return op(la, ra);
    }
};

```

### 1.6 Segment Tree 2D

Query:  $O(\log N \cdot \log M)$

Update:  $O(\log N \cdot \log M)$

```
template <typename T, auto op>
class SegTree {
private:
    T e;
    ll n, m;
    vector<vector<T>> seg;

public:
    SegTree(ll n, ll m, T e)
        : e(e), n(n), m(m), seg(2 * n, vector<T>(2 * m, e)) {}

    void assign(ll x, ll y, T v) {
        ll ny = y += m;
        for (x += n; x; x >>= 1, y = ny) {
            if (x >= n)
                seg[x][y] = v;
            else
                seg[x][y] = op(seg[2 * x][y], seg[2 * x + 1][y]);

            while (y >>= 1) seg[x][y] = op(seg[x][2 * y], seg[x][2 * y + 1]);
        }
    }

    T query(ll lx, ll rx, ll ly, ll ry) {
        ll ans = e, nx = rx + n, my = ry + m;

        for (lx += n, ly += m; lx <= ly; ++lx >>= 1, --ly >>= 1)
            for (rx = nx, ry = my; rx <= ry; ++rx >>= 1, --ry >>= 1) {
                if (lx & 1 and rx & 1) ans = op(ans, seg[lx][rx]);
                if (lx & 1 and !(ry & 1)) ans = op(ans, seg[lx][ry]);
                if (!(ly & 1) and rx & 1) ans = op(ans, seg[ly][rx]);
                if (!(ly & 1) and !(ry & 1)) ans = op(ans, seg[ly][ry]);
            }

        return ans;
    }
};
```

## 1.7 Union Find Disjoint Set (UFDS)

Uncomment the lines to recover which element belong to each set.

Time:  $\approx O(1)$  for everything.

```
class UFDS {
public:
    vi ps, size;
    // vector<unordered_set<int>> sts;

    UFDS(int N) : size(N + 1, 1), ps(N + 1), sts(N) {
        iota(ps.begin(), ps.end(), 0);
        // for (int i = 0; i < N; i++) sts[i].insert(i);
    }

    int find_set(int x) { return x == ps[x] ? x : (ps[x] = find_set(ps[x])); }

    bool same_set(int x, int y) { return find_set(x) == find_set(y); }
```

```
void union_set(int x, int y) {
    if (same_set(x, y)) return;

    int px = find_set(x);
    int py = find_set(y);

    if (size[px] < size[py]) swap(px, py);

    ps[py] = px;
    size[px] += size[py];
    // sts[px].merge(sts[py]);
}
};
```

## 2 Dynamic programming

### 2.1 Kadane

```
int kadane(const vi& xs) {
    vi s(xs.size());
    s[0] = xs[0];

    for (size_t i = 1; i < xs.size(); ++i) s[i] = max(xs[i], s[i - 1] + xs[i]);

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

### 2.2 Longest Increasing Subsequence (LIS)

Time:  $O(N \cdot \log N)$ .

```
int lis(vi const& a) {
    int n = a.size();
    const int INF = 1e9;
    vi d(n + 1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[l - 1] < a[i] && a[i] < d[l]) d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF) ans = l;
    }

    return ans;
}
```

## 3 Geometry

### 3.1 Convex Hull

Given a set of points find the smallest convex polygon that contains all the given points.

Time:  $O(N \cdot \log N)$

By default it removes the collinear points, set the boolean to true if you don't want that

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}

bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x - a.x) * (p0.x - a.x) + (p0.y - a.y) * (p0.y - a.y) <
                (p0.x - b.x) * (p0.x - b.x) + (p0.y - b.y) * (p0.y - b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size() - 1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin() + i + 1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 &&
            !cw(st[st.size() - 2], st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    a = st;
}
```

## 3.2 Point To Segment

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

```
double pt2segment(pdb A, pdb B, pdb E) {
    pdb AB = {B.fst - A.fst, B.snd - A.snd};
    pdb BE = {E.fst - B.fst, E.snd - B.snd};
    pdb AE = {E.fst - A.fst, E.snd - A.snd};
```

```
double AB_BE = AB.fst * BE.fst + AB.snd * BE.snd;
double AB_AE = AB.fst * AE.fst + AB.snd * AE.snd;
```

```
double ans;
if (AB_BE > 0) {
    double y = E.snd - B.snd;
    double x = E.fst - B.fst;
    ans = hypot(x, y);
} else if (AB_AE < 0) {
    double y = E.snd - A.snd;
    double x = E.fst - A.fst;
    ans = hypot(x, y);
} else {
    auto [x1, y1] = AB;
    auto [x2, y2] = AE;
    double mod = hypot(x1, y1);
    ans = abs(x1 * y2 - y1 * x2) / mod;
}

return ans;
}
```

## 4 Graphs

### 4.1 Articulation Points

```
int dfs_num[MAX], dfs_low[MAX];
vi adj[MAX];

int dfs_articulation_points(int u, int p, int& next, set<int>& points) {
    int children = 0;
    dfs_low[u] = dfs_num[u] = next++;

    for (auto v : adj[u])
        if (not dfs_num[v]) {
            ++children;

            dfs_articulation_points(v, u, next, points);

            if (dfs_low[v] >= dfs_num[u]) points.insert(u);

            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
        } else if (v != p)
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);

    return children;
}

set<int> articulation_points(int N) {
    memset(dfs_num, 0, (N + 1) * sizeof(int));
    memset(dfs_low, 0, (N + 1) * sizeof(int));

    set<int> points;

    for (int u = 1, next = 1; u <= N; ++u)
        if (not dfs_num[u]) {
            auto children = dfs_articulation_points(u, u, next, points);
```

```

        if (children == 1) points.erase(u);
    }

    return points;
}

```

## 4.2 Bellman Ford

Time:  $O(V \cdot E)$ . Returns the shortest path from  $s$  to all other nodes.

```

using edge = tuple<int, int, int>;

pair<vi, vi> bellman_ford(int s, int N, const vector<edge>& edges) {
    vi dist(N + 1, oo), pred(N + 1, oo);

    dist[s] = 0;
    pred[s] = s;

    for (int i = 1; i <= N - 1; i++)
        for (auto [u, v, w] : edges)
            if (dist[u] < oo and dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pred[v] = u;
            }

    return {dist, pred};
}

```

## 4.3 BFS 0/1

Time:  $O(V + E)$ .

```

vii adj[MAX];

vi bfs_01(int s, int N) {
    vi dist(N + 1, oo);
    dist[s] = 0;

    deque<int> q;
    q.emplace_back(s);

    while (not q.empty()) {
        auto u = q.front();
        q.pop_front();

        for (auto [v, w] : adj[u])
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                w == 0 ? q.emplace_front(v) : q.emplace_back(v);
            }
    }

    return dist;
}

```

## 4.4 Bridges

```

int dfs_num[MAX], dfs_low[MAX];
vi adj[MAX];

void dfs_bridge(int u, int p, int& next, vii& bridges) {
    dfs_low[u] = dfs_num[u] = next++;

    for (auto v : adj[u])
        if (not dfs_num[v]) {
            dfs_bridge(v, u, next, bridges);

            if (dfs_low[v] > dfs_num[u]) bridges.emplace_back(u, v);

            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
        } else if (v != p)
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);
    }

    vii bridges(int N) {
        memset(dfs_num, 0, (N + 1) * sizeof(int));
        memset(dfs_low, 0, (N + 1) * sizeof(int));

        vii bridges;

        for (int u = 1, next = 1; u <= N; ++u)
            if (not dfs_num[u]) dfs_bridge(u, u, next, bridges);

        return bridges;
    }
}

```

## 4.5 Negative Cycle Bellman Ford

Time:  $O(V \cdot E)$ . Detects whether there is a negative cycle in the graph using Bellman Ford.

```

using edge = tuple<int, int, int>;

bool has_negative_cycle(int s, int N, const vector<edge>& edges) {
    vi dist(N + 1, oo);
    dist[s] = 0;

    for (int i = 1; i <= N - 1; i++)
        for (auto [u, v, w] : edges)
            if (dist[u] < oo and dist[v] > dist[u] + w) dist[v] = dist[u] + w;

    for (auto [u, v, w] : edges)
        if (dist[u] < oo and dist[v] > dist[u] + w) return true;

    return false;
}

```

## 4.6 Negative Cycle Floyd Warshall

Time:  $O(n^3)$ . Detects whether there is a negative cycle in the graph using Floyd Warshall.

```

int dist[MAX][MAX];
vii adj[MAX];

```

```

bool has_negative_cycle(int N) {
    for (int u = 1; u <= N; ++u)
        for (int v = 1; v <= N; ++v) dist[u][v] = u == v ? 0 : oo;

    for (int u = 1; u <= N; ++u)
        for (auto [v, w] : adj[u]) dist[u][v] = w;

    for (int k = 1; k <= N; ++k)
        for (int u = 1; u <= N; ++u)
            for (int v = 1; v <= N; ++v)
                if (dist[u][k] < oo and dist[k][v] < oo)
                    dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v]);

    for (int i = 1; i <= N; ++i)
        if (dist[i][i] < 0) return true;

    return false;
}

```

## 4.7 Dijkstra

```

pair<vl, vl> Graph::dijkstra(ll src) {
    vl pd(this->N, LLONG_MAX), ds(this->N, LLONG_MAX);
    pd[src] = src;
    ds[src] = 0;

    set<pll> st;
    st.emplace(0, src);

    while (!st.empty()) {
        ll u = st.begin()->snd;
        ll wu = st.begin()->fst;
        st.erase(st.begin());

        if (wu != ds[u]) continue;
        for (auto& [v, w] : adj[u]) {
            if (ds[v] > ds[u] + w) {
                ds[v] = ds[u] + w;
                pd[v] = u;
                st.emplace(ds[v], v);
            }
        }
    }

    return {ds, pd};
}

```

## 4.8 Floyd Warshall

```

vii adj[MAX];

pair<vector<vi>, vector<vi>> floyd_warshall(int N) {
    vector<vi> dist(N + 1, vi(N + 1, oo));
    vector<vi> pred(N + 1, vi(N + 1, oo));

    for (int u = 1; u <= N; ++u) {

```

```

        dist[u][u] = 0;
        pred[u][u] = u;
    }

    for (int u = 1; u <= N; ++u)
        for (auto [v, w] : adj[u]) {
            dist[u][v] = w;
            pred[u][v] = u;
        }

    for (int k = 1; k <= N; ++k) {
        for (int u = 1; u <= N; ++u) {
            for (int v = 1; v <= N; ++v) {
                if (dist[u][k] < oo and dist[k][v] < oo and
                    dist[u][v] > dist[u][k] + dist[k][v]) {
                    dist[u][v] = dist[u][k] + dist[k][v];
                    pred[u][v] = pred[k][v];
                }
            }
        }
    }

    return {dist, pred};
}

```

## 4.9 Graph

```

class Graph {
private:
    ll N;
    bool undirected;
    vector<vll> adj;

public:
    Graph(ll N, bool is_undirected = true) {
        this->N = N;
        adj.resize(N);
        undirected = is_undirected;
    }

    void add(ll u, ll v, ll w) {
        adj[u].emplace_back(v, w);
        if (undirected) adj[v].emplace_back(u, w);
    }
};

```

## 4.10 TopSort - Kahn

Works only on Directed Acyclic Graphs (DAGs). For each edge (u,v), u comes before v in the ordering. If the task A is a prerequisite for task B, then A comes before B in the ordering. Time:  $O(E \cdot \log(v))$

```

unordered_set<int> in[MAX], out[MAX];

vi topological_sort(int N) {
    vi o;
    queue<int> q;

    for (int u = 1; u <= N; ++u)
        if (in[u].empty()) q.push(u);

```



```

while (not q.empty()) {
    auto u = q.front();
    q.pop();

    o.emplace_back(u);

    for (auto v : out[u]) {
        in[v].erase(u);

        if (in[v].empty()) q.push(v);
    }
}

return (int)o.size() == N ? o : vi{};
}

```

## 4.11 Kruskal

Time:  $O(e \cdot \log(v))$

```

using edge = tuple<int, int, int>;

int kruskal(int N, vector<edge>& es) {
    sort(es.begin(), es.end());

    int cost = 0;
    UnionFind udfs(N);

    for (auto [w, u, v] : es) {
        if (not udfs.same_set(u, v)) {
            cost += w;
            udfs.union_set(u, v);
        }
    }

    return cost;
}

```

## 4.12 Minimax

A MST minimizes the maximum weight between the edges in any spanning tree. Time:  $O(e \cdot \log(v))$

```

vii adj[MAX];

int minimax(int u, int N) {
    set<int> C;
    C.insert(u);

    priority_queue<ii, vii, greater<ii>> pq;

    for (auto [v, w] : adj[u]) pq.push(ii(w, v));

    int minmax = -oo;

    while ((int)C.size() < N) {
        int v, w;

```

```

        do {
            w = pq.top().first, v = pq.top().second;
            pq.pop();
        } while (C.count(v));

        minmax = max(minmax, w);
        C.insert(v);

        for (auto [s, p] : adj[v]) pq.push(ii(p, s));
    }

    return minmax;
}

```

## 4.13 MSF

Minimum Spanning Forest - a forest of trees of length  $k$  that connects all vertices in a graph with minimum total weight. Time:  $O(e \cdot \log(v))$

```

using edge = tuple<int, int, int>;

int msf(int k, int N, vector<edge>& es) {
    sort(es.begin(), es.end());

    int cost = 0, cc = N;
    UnionFind udfs(N);

    for (auto [w, u, v] : es) {
        if (not udfs.same_set(u, v)) {
            cost += w;
            udfs.union_set(u, v);

            if (--cc == k) return cost;
        }
    }

    return cost;
}

```

## 4.14 Minimum Spanning Graph (MSG)

Given some obligatory edges  $es$ , find a minimum spanning graph that contains them. Time:  $O(e \cdot \log(v))$

```

using edge = tuple<int, int, int>;

const int MAX{100010};

vector<ii> adj[MAX];

int msg(int N, const vector<edge>& es) {
    set<int> C;
    priority_queue<ii, vii, greater<ii>> pq;
    int cost = 0;

    for (auto [u, v, w] : es) {
        cost += w;

```

```

C.insert(u);
C.insert(v);

for (auto [r, s] : adj[u]) pq.push(ii(s, r));

for (auto [r, s] : adj[v]) pq.push(ii(s, r));
}

while ((int)C.size() < N) {
    int v, w;

    do {
        w = pq.top().first, v = pq.top().second;
        pq.pop();
    } while (C.count(v));

    cost += w;
    C.insert(v);

    for (auto [s, p] : adj[v]) pq.push(ii(p, s));
}

return cost;
}

```

## 4.15 Prim

A node  $u$  is chosen to start a connected component. Time:  $O(e \cdot \log(v))$

```

const int MAX{100010};

vector<ii> adj[MAX];

int prim(int u, int N) {
    set<int> C;
    C.insert(u);

    priority_queue<ii, vector<ii>, greater<ii>> pq;

    for (auto [v, w] : adj[u]) pq.push(ii(w, v));

    int mst = 0;

    while ((int)C.size() < N) {
        int v, w;

        do {
            w = pq.top().first, v = pq.top().second;
            pq.pop();
        } while (C.count(v));

        mst += w;
        C.insert(v);

        for (auto [s, p] : adj[v]) pq.push(ii(p, s));
    }
}

```

```

return mst;
}

```

## 4.16 Retrieve Path 2d

```

vll Graph::retrieve_path_2d(ll src, ll trg, const vector<vl>& pred) {
    vll p;

    do {
        p.emplace_back(pred[src][trg], trg);
        trg = pred[src][trg];
    } while (trg != src);

    reverse(all(p));

    return p;
}

```

## 4.17 Retrieve Path

```

vll Graph::retrieve_path(ll src, ll trg, const vl& pred) {
    vll p;

    do {
        p.emplace_back(pred[trg], trg);
        trg = pred[trg];
    } while (trg != src);

    reverse(all(p));

    return p;
}

```

## 4.18 Second Best MST

Time:  $O(v \cdot e)$

```

using edge = tuple<int, int, int>;

pair<int, vi> kruskal(int N, vector<edge>& es, int blocked = -1) {
    vi mst;
    UnionFind udfs(N);
    int cost = 0;

    for (int i = 0; i < (int)es.size(); ++i) {
        auto [w, u, v] = es[i];

        if (i != blocked and not udfs.same_set(u, v)) {
            cost += w;
            udfs.union_set(u, v);
            mst.emplace_back(i);
        }
    }

    return {(int)mst.size() == N - 1 ? cost : oo, mst};
}

```

```
int second_best(int N, vector<edge>& es) {
    sort(es.begin(), es.end());

    auto [_ , mst] = kruskal(N, es);
    int best = oo;

    for (auto blocked : mst) {
        auto [cost, __] = kruskal(N, es, blocked);
        best = min(best, cost);
    }

    return best;
}
```

## 4.19 TopSort - Tarjan

Works only on Directed Acyclic Graphs (DAGs). For each edge (u,v), u comes before v in the ordering. If the task A is a prerequisite for task B, then A comes before B in the ordering. Time:  $O(V + E)$

```
enum State { NOT_FOUND, FOUND, PROCESSED };

vi adj[MAX];

bool dfs(int u, vi& o, vi& state) {
    if (state[u] == PROCESSED) return true;

    if (state[u] == FOUND) return false;

    state[u] = FOUND;

    for (auto v : adj[u])
        if (not dfs(v, o, state)) return false;

    state[u] = PROCESSED;
    o.emplace_back(u);

    return true;
}

vi topological_sort(int N) {
    vi o, state(N + 1, NOT_FOUND);

    for (int u = 1; u <= N; ++u)
        if (state[u] == NOT_FOUND and not dfs(u, o, state)) return {};

    reverse(o.begin(), o.end());

    return o;
}
```

# 5 Math

## 5.1 Binomial

```
ll binom(ll n, ll k) {
```

```
    if (k > n) return 0;
    vll dp(k + 1, 0);
    dp[0] = 1;
    for (ll i = 1; i <= n; i++)
        for (ll j = k; j > 0; j--) dp[j] = dp[j] + dp[j - 1];
    return dp[k];
}
```

## 5.2 Count Divisors Range

```
vll divisors(MAX, 0);
void count_divisors_range() {
    for (ll i = 1; i <= MAX; i++) {
        for (ll j = 1; j * i <= MAX; j++) divisors[i * j]++;
    }
}
```

## 5.3 Count Divisors

```
ll count_divisors(ll num) {
    ll count = 1;
    for (int i = 2; (ll)i * i <= num; i++) {
        if (num % i == 0) {
            int e = 0;
            do {
                e++;
                num /= i;
            } while (num % i == 0);
            count *= e + 1;
        }
    }
    if (num > 1) {
        count *= 2;
    }
    return count;
}
```

## 5.4 Factorization With Sieve

```
map<ll, ll> factorization_with_sieve(ll n, const vl& primes) {
    map<ll, ll> fact;

    for (ll d : primes) {
        if (d * d > n) break;

        ll k = 0;
        while (n % d == 0) {
            k++;
            n /= d;
        }

        if (k) fact[d] = k;
    }

    if (n > 1) fact[n] = 1;
    return fact;
}
```

## 5.5 Factorization

```
map<ll, ll> factorization(ll n) {
    map<ll, ll> ans;
    for (ll i = 2; i * i <= n; i++) {
        ll count = 0;
        for (; n % i == 0; count++, n /= i)
            ;
        if (count) ans[i] = count;
    }
    if (n > 1) ans[n]++;
    return ans;
}
```

## 5.6 Fast Doubling - Fibonacci

The Doubling Method can be seen as an improvement to the matrix exponentiation method to find the  $N$ -th Fibonacci number.  
Time:  $O(\log N)$ .

```
template <typename T>
class FastDoubling {
public:
    vector<T> sts;
    T a, b, c, d;
    int mod;

    FastDoubling(int mod = 1e9 + 7) : sts(2), mod(mod) {}

    T fib(T x) {
        fill(all(sts), 0);
        a = 0, b = 0, c = 0, d = 0;
        fast_doubling(x, sts);
        return sts[0];
    }

    void fast_doubling(T n, vector<T>& res) {
        if (n == 0) {
            res[0] = 0;
            res[1] = 1;
            return;
        }
        fast_doubling(n >> 1, res);

        a = res[0];
        b = res[1];
        c = (b << 1) - a;

        if (c < 0) c += mod;

        c = (a * c) % mod;
        d = (a * a + b * b) % mod;
        if (n & 1) {
            res[0] = d;
            res[1] = c + d;
        } else {
            res[0] = c;
            res[1] = d;
        }
    }
}
```

```
    }
};
```

## 5.7 Fast Exp Iterative

```
ll fast_exp_it(ll a, ll n, ll mod = LLONG_MAX) {
    a %= mod;
    ll res = 1;

    while (n) {
        if (n & 1) (res *= a) %= mod;

        (a *= a) %= mod;
        n >>= 1;
    }

    return res;
}
```

## 5.8 Fast Exp

```
ll fast_exp(ll a, ll n, ll mod = LLONG_MAX) {
    if (n == 0) return 1;
    if (n == 1) return a;

    ll x = fast_exp(a, n / 2, mod) % mod;

    return ((x * x) % mod * (n & 1 ? a : 1ll)) % mod;
}
```

## 5.9 GCD

The Euclidean algorithm allows to find the greatest common divisor of two numbers  $a$  and  $b$  in  $O(\log \cdot \min(a, b))$ .

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

## 5.10 Integer Mod

```
const ll INF = 1e18;
const ll mod = 998244353;
template <ll MOD = mod>

struct Modular {
    ll value;
    static const ll MOD_value = MOD;

    Modular(ll v = 0) {
        value = v % MOD;
        if (value < 0) value += MOD;
    }
    Modular(ll a, ll b) : value(0) {
        *this += a;
        *this /= b;
    }
}
```

```

Modular& operator+=(Modular const& b) {
    value += b.value;
    if (value >= MOD) value -= MOD;
    return *this;
}
Modular& operator--(Modular const& b) {
    value -= b.value;
    if (value < 0) value += MOD;
    return *this;
}
Modular& operator*=(Modular const& b) {
    value = (ll)value * b.value % MOD;
    return *this;
}

friend Modular mexp(Modular a, ll e) {
    Modular res = 1;
    while (e) {
        if (e & 1) res *= a;
        a *= a;
        e >>= 1;
    }
    return res;
}
friend Modular inverse(Modular a) { return mexp(a, MOD - 2); }

Modular& operator/=(Modular const& b) { return *this *= inverse(b); }
friend Modular operator+(Modular a, Modular const b) { return a += b; }
Modular operator++(int) { return this->value = (this->value + 1) % MOD; }
Modular operator++() { return this->value = (this->value + 1) % MOD; }
friend Modular operator-(Modular a, Modular const b) { return a -= b; }
friend Modular operator-(Modular const a) { return 0 - a; }
Modular operator--(int) {
    return this->value = (this->value - 1 + MOD) % MOD;
}

Modular operator--() { return this->value = (this->value - 1 + MOD) % MOD; }
friend Modular operator*(Modular a, Modular const b) { return a *= b; }
friend Modular operator/(Modular a, Modular const b) { return a /= b; }
friend std::ostream& operator<<(std::ostream& os, Modular const& a) {
    return os << a.value;
}

friend bool operator==(Modular const& a, Modular const& b) {
    return a.value == b.value;
}
friend bool operator!=(Modular const& a, Modular const& b) {
    return a.value != b.value;
}
};

```

## 5.11 Is prime

$O(\sqrt{N})$

```

bool isprime(ll n) {
    if (n < 2) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;
    for (ll i = 3; i * i < n; i += 2)

```

```

        if (n % i == 0) return false;
    return true;
}

```

## 5.12 LCM

Calculating the least common multiple (commonly denoted LCM) can be reduced to calculating the GCD with the following simple formula:  $\text{lcm}(a, b) = (a \cdot b) / \text{gcd}(a, b)$

Thus, LCM can be calculated using the Euclidean algorithm with the same time complexity:

```

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

```

## 5.13 Euler phi $\varphi(n)$

Computes the number of positive integers less than  $n$  that are co-primes with  $n$ , in  $O(\sqrt{N})$ .

```

ll phi(ll n) {
    if (n == 1) return 1;

    auto fs = factorization(n);
    auto res = n;

    for (auto [p, k] : fs) {
        res /= p;
        res *= (p - 1);
    }

    return res;
}

```

## 5.14 Sieve

```

vl sieve(ll N) {
    bitset<MAX + 1> sieve;
    vl ps{2, 3};
    sieve.set();

    for (ll i = 5, step = 2; i <= N; i += step, step = 6 - step) {
        if (sieve[i]) {
            ps.push_back(i);

            for (ll j = i * i; j <= N; j += 2 * i) sieve[j] = false;
        }
    }
    return ps;
}

```

## 5.15 Sum Divisors

```

ll sum_divisors(ll num) {
    ll result = 1;

    for (int i = 2; (ll)i * i <= num; i++) {
        if (num % i == 0) {
            int e = 0;
            do {
                e++;

```

```

    num /= i;
} while (num % i == 0);

ll sum = 0, pow = 1;
do {
    sum += pow;
    pow *= i;
} while (e-- > 0);
result *= sum;
}
}
if (num > 1) {
    result *= (1 + num);
}
return result;
}

```

## 5.16 Sum of difference

Function to calculate sum of absolute difference of all pairs in array:  $\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N |A_i - A_j|$

```

ll sum_of_difference(vl& arr, ll n) {
    sort(all(arr));

    ll sum = 0;
    for (ll i = 0; i < n; i++) {
        sum += i * arr[i] - (n - 1 - i) * arr[i];
    }

    return sum;
}

```

# 6 Problems

## 6.1 Kth Digit String (CSES)

Time:  $O(\log_{10} K)$ .

Space:  $O(1)$ .

```

ll kth_digit_string(ll k) {
    if (k < 10) return k;

    ll c = 180, i = 2, u = 10, r = 0, ans = -1, m;
    for (k -= 9; k > c; i++, u *= 10) {
        k -= c;
        c /= i;
        c *= 10 * (i + 1);
    }

    if ((m = k % i))
        r++;
    else
        m = i;

    ll tmp = (k / i) + r + u - 1;
    for (m = i + 1 - m; m--; tmp /= 10) ans = tmp % 10;
}

```

```

    return ans;
}

```

# 7 Strings

## 7.1 Edit Distance

Returns the minimum number of operations (insert, delete, replace) to transform string  $a$  into string  $b$ .

Time:  $O(M * N)$

```

int min_value(int x, int y, int z) { return min(min(x, y), z); }

int edit_distance(string str1, string str2) {
    int n = (int)str1.size(), m = (int)str2.size();
    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++)
            if (i == 0)
                dp[i][j] = j;
            else if (j == 0)
                dp[i][j] = i;
            else if (str1[i - 1] == str2[j - 1])
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = 1 + min_value(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]);

    return dp[m][n];
}

```

## 7.2 Manacher

Given string  $s$  with length  $n$ . Find all the pairs  $(i, j)$  such that substring  $s[i \dots j]$  is a palindrome. String  $t$  is a palindrome when  $t = t_{rev}$  ( $t_{rev}$  is a reversed string for  $t$ ).

Time:  $O(N)$

```

vi manacher(string s) {
    string t;
    for (auto c : s) t += string("#") + c;
    t = t + '#';

    int n = t.size();
    t = "$" + t + "^";

    vi p(n + 2);
    int l = 1, r = 1;
    for (int i = 1; i <= n; i++) {
        p[i] = max(0, min(r - i, p[l + (r - i)]));
        while (t[i - p[i]] == t[i + p[i]]) p[i]++;
        if (i + p[i] > r) {
            l = i - p[i], r = i + p[i];
        }
        p[i]--;
    }
}

```

```

    return vi(begin(p) + 1, end(p) - 1);
}

```

## 7.3 Rabin Karp

```

vector<int> rabin_karp(string const& s, string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();

    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++) p_pow[i] = (p_pow[i - 1] * p) %
        m;

    vector<long long> h(T + 1, 0);
    for (int i = 0; i < T; i++)
        h[i + 1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;
    long long h_s = 0;
    for (int i = 0; i < S; i++) h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;

    vector<int> occurrences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i + S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m) occurrences.push_back(i);
    }

    return occurrences;
}

```

## 8 Trees

### 8.1 LCA Binary Lifting (CP Algo)

The algorithm described will need  $O(N \cdot \log N)$  for preprocessing the tree, and then  $O(\log N)$  for each LCA query.

```

ll n, l;
vector<ll> adj[MAX];

ll timer;
vector<ll> tin, tout;
vector<vector<ll>> up;

void dfs(ll v, ll p) {
    tin[v] = ++timer;
    up[v][0] = p;
    for (ll i = 1; i <= l; ++i) up[v][i] = up[up[v][i - 1]][i - 1];

    for (ll u : adj[v]) {
        if (u != p) dfs(u, v);
    }

    tout[v] = ++timer;
}

```

```

bool is_ancestor(ll u, ll v) { return tin[u] <= tin[v] && tout[u] >= tout[v];
}

```

```

ll lca(ll u, ll v) {
    if (is_ancestor(u, v)) return u;
    if (is_ancestor(v, u)) return v;
    for (ll i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v)) u = up[u][i];
    }
    return up[u][0];
}

```

```

void preprocess(ll root) {
    tin.resize(n);
    tout.resize(n);
    timer = 0;
    l = ceil(log2(n));
    up.assign(n, vector<ll>(l + 1));
    dfs(root, root);
}

```

### 8.2 LCA SegTree (CP Algo)

The algorithm can answer each query in  $O(\log N)$  with preprocessing in  $O(N)$  time.

```

struct LCA {
    vector<ll> height, euler, first, segtree;
    vector<bool> visited;
    ll n;

    LCA(vector<vector<ll>>& adj, ll root = 0) {
        n = adj.size();
        height.resize(n);
        first.resize(n);
        euler.reserve(n * 2);
        visited.assign(n, false);
        dfs(adj, root);
        ll m = euler.size();
        segtree.resize(m * 4);
        build(1, 0, m - 1);
    }

    void dfs(vector<vector<ll>>& adj, ll node, ll h = 0) {
        visited[node] = true;
        height[node] = h;
        first[node] = euler.size();
        euler.push_back(node);
        for (auto to : adj[node]) {
            if (!visited[to]) {
                dfs(adj, to, h + 1);
                euler.push_back(node);
            }
        }
    }

    void build(ll node, ll b, ll e) {
        if (b == e) {
            segtree[node] = euler[b];

```

```

} else {
    ll mid = (b + e) / 2;
    build(node << 1, b, mid);
    build(node << 1 | 1, mid + 1, e);
    ll l = segtree[node << 1], r = segtree[node << 1 | 1];
    segtree[node] = (height[l] < height[r]) ? l : r;
}
}

ll query(ll node, ll b, ll e, ll L, ll R) {
    if (b > R || e < L) return -1;
    if (b >= L && e <= R) return segtree[node];
    ll mid = (b + e) >> 1;

    ll left = query(node << 1, b, mid, L, R);
    ll right = query(node << 1 | 1, mid + 1, e, L, R);
    if (left == -1) return right;
    if (right == -1) return left;
    return height[left] < height[right] ? left : right;
}

ll lca(ll u, ll v) {
    ll left = first[u], right = first[v];
    if (left > right) swap(left, right);
    return query(1, 0, euler.size() - 1, left, right);
}
};

```

### 8.3 LCA Sparse Table

The algorithm described will need  $O(N)$  for preprocessing, and then  $O(1)$  for each LCA query.

0 indexed!

```

typedef vector<vl> vl2d;
#define all(a) a.begin(), a.end()
#define len(x) (int)x.size()

template <typename T>
struct SparseTable {
    vector<T> v;
    ll n;
    static const ll b = 30;
    vl mask, t;

    ll op(ll x, ll y) { return v[x] < v[y] ? x : y; }
    ll msb(ll x) { return __builtin_clz(1) - __builtin_clz(x); }
    SparseTable() {}
    SparseTable(const vector<T>& v_) : v(v_), n(v.size()), mask(n), t(n) {
        for (ll i = 0, at = 0; i < n; mask[i++] = at | = 1) {
            at = (at << 1) & ((1 << b) - 1);
            while (at and op(i, i - msb(at & -at)) == i) at ^= at & -at;
        }
        for (ll i = 0; i < n / b; i++)
            t[i] = b * i + b - 1 - msb(mask[b * i + b - 1]);
        for (ll j = 1; (1 << j) <= n / b; j++)
            for (ll i = 0; i + (1 << j) <= n / b; i++)
                t[n / b * j + i] =
                    op(t[n / b * (j - 1) + i], t[n / b * (j - 1) + i + (1 << (j - 1))]);
    }
};

```

```

}
ll small(ll r, ll sz = b) { return r - msb(mask[r] & ((1 << sz) - 1)); }
T query(ll l, ll r) {
    if (r - l + 1 <= b) return small(r, r - l + 1);
    ll ans = op(small(l + b - 1), small(r));
    ll x = l / b + 1, y = r / b - 1;
    if (x <= y) {
        ll j = msb(y - x + 1);
        ans = op(ans, op(t[n / b * j + x], t[n / b * j + y - (1 << j) + 1]));
    }
    return ans;
}
};

struct LCA {
    SparseTable<ll> st;
    ll n;
    vl v, pos, dep;

    LCA(const vl2d& g, ll root) : n(len(g)), pos(n) {
        dfs(root, 0, -1, g);
        st = SparseTable<ll>(vector<ll>(all(dep)));
    }

    void dfs(ll i, ll d, ll p, const vl2d& g) {
        v.emplace_back(len(dep)) = i, pos[i] = len(dep), dep.emplace_back(d);
        for (auto j : g[i])
            if (j != p) {
                dfs(j, d + 1, i, g);
                v.emplace_back(len(dep)) = i, dep.emplace_back(d);
            }
    }

    ll lca(ll a, ll b) {
        ll l = min(pos[a], pos[b]);
        ll r = max(pos[a], pos[b]);
        return v[st.query(l, r)];
    }

    ll dist(ll a, ll b) {
        return dep[pos[a]] + dep[pos[b]] - 2 * dep[pos[lca(a, b)]];
    }
};

```

### 8.4 Tree Flatten

```

vll tree_flatten(ll root) {
    vl pre;
    pre.reserve(N);

    vll flat(N);
    ll timer = -1;
    auto dfs = [&](auto&& self, ll u, ll p) -> void {
        timer++;
        pre.push_back(u);
        for (auto [v, w] : adj[u])
            if (v != p) {
                self(self, v, u);
            }
    };
    dfs(root, -1);
    return flat;
}

```



```

    }
    flat[u].second = timer;
};
dfs(dfs, root, -1);
for (ll i = 0; i < (ll)N; i++) flat[pre[i]].first = i;
return flat;
}

```

## 8.5 Tree Isomorph

Checks whether two tree are isomorph. The function *thash()* returns the hash of the tree (using centroids as special vertices). Two trees are isomorph if their hash are the same.

```
map<vector<int>, int> mphash;
```

```

struct tree {
    int n;
    vector<vector<int>> g;
    vector<int> sz, cs;

    tree(int n_) : n(n_), g(n_), sz(n_) {}

    void dfs_centroid(int v, int p) {
        sz[v] = 1;
        bool cent = true;
        for (int u : g[v])
            if (u != p) {
                dfs_centroid(u, v), sz[v] += sz[u];
                if (sz[u] > n / 2) cent = false;
            }
        if (cent and n - sz[v] <= n / 2) cs.push_back(v);
    }
    int fhash(int v, int p) {
        vector<int> h;
        for (int u : g[v])
            if (u != p) h.push_back(fhash(u, v));
        sort(h.begin(), h.end());
        if (!mphash.count(h)) mphash[h] = mphash.size();
        return mphash[h];
    }
    ll thash() {
        cs.clear();
        dfs_centroid(0, -1);
        if (cs.size() == 1) return fhash(cs[0], -1);
        ll h1 = fhash(cs[0], cs[1]), h2 = fhash(cs[1], cs[0]);
        return (min(h1, h2) << 30) + max(h1, h2);
    }
    void add(int a, int b) {
        g[a].emplace_back(b);
        g[b].emplace_back(a);
    }
};

```

## 9 Settings and macros

### 9.1 short-macro.cpp

```

#include <bits/stdc++.h>

using namespace std;

#ifdef DEBUG
#include "../settings-and-macros/debug.cpp"
#else
#define dbg(...)
#endif

typedef long long ll;
typedef pair<int, int> ii;

#define all(x) x.begin(), x.end()
#define vin(vt) for (auto &e : vt) cin >> e

auto solve() { }

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

    ll t = 1;
    //cin >> t;

    while (t--) solve();

    return 0;
}

```

### 9.2 macro.cpp

```

#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
#define ordered_set tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update>

using namespace std;

#ifdef DEBUG
#include "../settings-and-macros/debug.cpp"
#else
#define dbg(...)
#endif

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef vector<int> vi;
typedef vector<ll> vl;
typedef vector<pii> vii;
typedef vector<pll> vll;

#define fst first
#define snd second

```

```
#define all(x) x.begin(), x.end()
#define len(vt) (int)vt.size()
#define vin(vt) for (auto &e : vt) cin >> e
#define LSOne(S) ((S) & -(S))
#define MSOne(S) (1ull << (63 - __builtin_clzll(S)))
#define fastio ios_base::sync_with_stdio(0); \
                cin.tie(0); \
                cout.tie(0)

const vii dir4 {{1,0},{-1,0},{0,1},{0,-1}};

auto solve() { }

int main() {
    fastio;

    ll t = 1;
    //cin >> t;

    while (t--) solve();

    return 0;
}
```

10 Theoretical guide

10.1 Modular Multiplicative Inverse

A modular multiplicative inverse of an integer  $a$  is an integer  $x$  such that  $a \cdot x$  is congruent to 1 modular some modulus  $m$ . To write it in a formal way:

$$a \cdot x \equiv 1 \pmod m.$$

Euler’s theorem, which states that the following congruence is true if  $a$  and  $m$  are co-primes:

$$a^{\phi(m)} \equiv 1 \pmod m$$

Multiply both sides of the above equations by  $a^{-1}$ , and we get:

- For an arbitrary (but coprime) modulus  $m$ :  $a^{\phi(m)-1} \equiv a^{-1} \pmod m$
- For a prime modulus  $m$ :  $a^{m-2} \equiv a^{-1} \pmod m$

From these results, we can easily find the modular inverse using the binary exponentiation algorithm, which works in  $O(\log m)$  time.

10.2 Exponent With Module

If  $a$  and  $m$  are coprime, then

$$a^n \equiv a^{n \pmod{\phi(m)}} \pmod m$$

Generally, if  $n \geq \log_2 m$ , then

$$a^n \equiv a^{\phi(m)+[n \pmod{\phi(m)}]} \pmod m$$

10.3 Notable Series

1. Sum of the first  $n$  naturals:

$$S_n = \sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

2. Sum of the squares of the first  $n$  naturals:

$$S_n = \sum_{i=1}^n i^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

3. Sum of the cubes of the first natural  $n$ :

$$S_n = \sum_{i=1}^n i^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \left[ \frac{n(n+1)}{2} \right]^2$$

4. Sum of the first  $n$  odd numbers:

$$S_n = \sum_{i=1}^n 2i - 1 = 1 + 3 + 5 + \dots + (2n - 1) = n^2$$