

Can Tho University

CTU.Stresstesters

Quang-Minh Tran
Cong-Huan Tran
Minh-Thien Nguyen

The ICPC 2024 Vietnam Central Provincial Contest

October 19-20, 2024

Contents

1 Contest	2	4.5 Suffix array	8	7.4 Stirling numbers of the second kind	13
1.1 Template	2	4.6 Suffix array slow	9	7.5 Derangements	13
1.2 Java	2	4.7 Manacher's algorithm	9		
2 Data structures	2	4.8 Trie	9	8 Geometry	14
2.1 RMQ	2	4.9 Aho-Corasick	9	8.1 Fundamentals	14
2.2 Ordered set/map	2	4.10 Hashing	10	8.2 KD tree	15
2.3 Dsu	2	4.11 Minimum rotation	10		
2.4 Dsu with rollback	2	5 Numerical	10	9 Linear algebra	16
2.5 MinQueue	3	5.1 Fast Fourier transform	10	9.1 Gauss elimination	16
2.6 Binary trie	3			9.2 Gauss determinant	16
2.7 Segment tree	3	6 Number Theory	10	9.3 Bareiss determinant	16
2.8 Efficient segment tree	4	6.1 Euler's totient function	10		
2.9 Persistent lazy segment tree	4	6.2 Mobius function	11	10 Graph	16
2.10 Lichao tree	4	6.3 Primes	11	10.1 Bellman-Ford algorithm	16
2.11 Old driver tree (Chtholly tree)	4	6.4 Wilson's theorem	11	10.2 Articulation point and Bridge	17
2.12 Disjoint sparse table	5	6.5 Zeckendorf's theorem	11	10.3 Topo sort	17
2.13 Fenwick tree	5	6.6 Chicken McNugget theorem	11	10.4 Strongly connected components	17
2.14 Treap	5	6.7 Bitwise operation	11	10.5 K-th smallest shortest path	17
2.15 Splay tree	6	6.8 Modmul	11	10.6 Eulerian path	18
2.16 Line container	6	6.9 Miller-Rabin	12	10.7 Network flow	18
2.17 Wavelet matrix	7	6.10 Pollard's rho algorithm	12	10.8 Trees	18
		6.11 Segment divisor sieve	12	10.9 2-SAT	20
3 Mathematics	7	6.12 Linear sieve	12	10.10 Manhattan MST	20
3.1 Trigonometry	7	6.13 Bitset sieve	12	10.11 Matching	20
3.2 Sums	8	6.14 Block sieve	12		
3.3 Pythagorean triple	8	6.15 Sqrt mod	13	11 Misc.	21
		6.16 Extended Euclidean	13	11.1 Ternary search	21
4 String	8			11.2 Gray code	21
4.1 Prefix function	8	7 Combinatorics	13	11.3 Matrix	21
4.2 Z function	8	7.1 Catalan numbers	13	11.4 K-th order statistic	21
4.3 Counting occurrences of each prefix	8	7.2 Fibonacci numbers	13	11.5 LIS	22
4.4 Knuth-Morris-Pratt algorithm	8	7.3 Stirling numbers of the first kind	13	11.6 Others	22

Contest

1.1 Template

template.h, 19 lines

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

#ifdef LOCAL
#include "cp/debug.h"
#else
#define debug(...)
#endif

mt19937 rng(
    chrono::steady_clock::now().time_since_epoch().count());

int main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    // freopen("input.txt", "r", stdin);
    // freopen("output.txt", "w", stdout);

    return 0;
}
```

1.2 Java

template.java, 50 lines

```
import java.io.BufferedReader;
import java.util.StringTokenizer;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        FastScanner fs = new FastScanner();
        PrintWriter out = new PrintWriter(System.out);
        int n = fs.nextInt();
        out.println(n);
        out.close(); // don't forget this line.
    }

    static class FastScanner {
        BufferedReader br;
        StringTokenizer st;
        public FastScanner() {
            br = new BufferedReader(new
                InputStreamReader(System.in));
            st = null;
        }
        public String next() {
            while (st == null || st.hasMoreTokens() ==
                false) {
                try {
                    st = new
                        StringTokenizer(br.readLine());
                }
                catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
            return st.nextToken();
        }

        public int nextInt() {
            return Integer.parseInt(next());
        }
    }
}
```

```

    }

    public long nextLong() {
        return Long.parseLong(next());
    }

    public double nextDouble() {
        return Double.parseDouble(next());
    }
}
}
```

Data structures

2.1 RMQ

Description: range minimum queries on a static array.

Time: $O(N \log N), O(1)$.

rmq.h, 24 lines

```
template<typename T> struct RMQ {
    int n;
    vector<vector<T>> rmq;
    RMQ() {}
    RMQ(const vector<T>& arr) { build(arr); }
    void build(const vector<T>& arr) {
        n = (int) arr.size();
        int max_log = __lg(n) + 1;
        rmq.resize(max_log);
        rmq[0] = arr;
        for (int j = 1; j < max_log; ++j) {
            rmq[j].resize(n - (1 << j) + 1);
            for (int i = 0; i + (1 << j) - 1 < n; ++i) {
                rmq[j][i] = min(
                    rmq[j - 1][i], rmq[j - 1][i + (1 << (j - 1))]);
            }
        }
    }
    T get(int l, int r) {
        assert(0 <= l && l <= r && r < n);
        int i = __lg(r - l + 1);
        return min(rmq[i][l], rmq[i][r - (1 << i) + 1]);
    }
};
```

2.2 Ordered set/map

ordered_set.h, 25 lines

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

template<typename K, typename V, typename comp = less<K>>
using ordered_map = tree<K, V, comp, rb_tree_tag,
    tree_order_statistics_node_update>;

template<typename K, typename comp = less<K>>
using ordered_set = ordered_map<K, null_type, comp>;

const int INF = 0x3f3f3f3f;
void example() {
    vector<int> nums = {1, 2, 3, 5, 10};
    ordered_set<int> st(nums.begin(), nums.end());

    cout << *st.find_by_order(0) << '\n'; // 1
    assert(st.find_by_order(-INF) == st.end());
    assert(st.find_by_order(INF) == st.end());

    cout << st.order_of_key(2) << '\n'; // 1
    cout << st.order_of_key(4) << '\n'; // 3
    cout << st.order_of_key(9) << '\n'; // 4
}
```

```
cout << st.order_of_key(-INF) << '\n'; // 0
cout << st.order_of_key(INF) << '\n'; // 5
}
```

2.3 Dsu

dsu.h, 39 lines

```
struct Dsu {
    int n;
    vector<int> par, sz;
    Dsu(int _n): n(_n), par(n), sz(n) { init(); }
    void init() {
        for (int i = 0; i < n; ++i) { par[i] = i, sz[i] = 1; }
    }
    int find(int v) {
        // finding leader/parent of set that contains the
        // element v. with {path compression optimization}.
        while (v != par[v]) { v = par[v] = par[par[v]]; }
        return v;
    }
    bool unite(int u, int v) {
        u = find(u);
        v = find(v);
        if (u == v) { return false; }
        if (sz[u] < sz[v]) { swap(u, v); }
        par[v] = u;
        sz[u] += sz[v];
        return true;
    }
    vector<vector<int>> groups() {
        // returns the list of the "list of the vertices in a
        // connected component".
        vector<int> leader(n);
        for (int i = 0; i < n; ++i) { leader[i] = find(i); }
        vector<int> id(n, -1);
        int count = 0;
        for (int i = 0; i < n; ++i) {
            if (id[leader[i]] == -1) { id[leader[i]] = count++; }
        }
        vector<vector<int>> result(count);
        for (int i = 0; i < n; ++i) {
            result[id[leader[i]]].emplace_back(i);
        }
        return result;
    }
};
```

2.4 Dsu with rollback

Description: a DSU with rollback operation, allow rollbacking to the previous snapshot.

Time: $O(\log N)$ per operation and $O(k)$ for rollback where k is the distance between snapshot to restore and the current snapshot.

dsu_with_rollback.h, 33 lines

```
struct DsuWithRollback {
    int n, comp;
    vector<int> par, rank;
    vector<tuple<int, int, int, int>> history;
    DsuWithRollback() {}
    DsuWithRollback(int _n): n(_n), comp(n), par(n),
        rank(n) {
        iota(par.begin(), par.end(), 0);
    }
    int find(int v) {
        while (v != par[v]) { v = par[v]; }
        return v;
    }
}
```

```

bool unite(int u, int v) {
    u = find(u), v = find(v);
    if (u == v) { return false; }
    --comp;
    if (rank[u] < rank[v]) { swap(u, v); }
    history.emplace_back(u, rank[u], v, rank[v]);
    par[v] = u;
    if (rank[u] == rank[v]) { ++rank[u]; }
    return true;
}

int snapshot() { return history.size(); }
void rollback(int until) {
    while (snapshot() > until) {
        auto [u, rank_u, v, rank_v] = history.back();
        history.pop_back();
        ++comp;
        par[u] = u, par[v] = v;
        rank[u] = rank_u, rank[v] = rank_v;
    }
}
};

```

2.5 MinQueue

Description: acts like normal std::queue except it supports get minimum value in current queue.

min_queue.h, 35 lines

```

template<typename T> struct MinQueue {
    vector<T> vals;
    int ptr = 0;
    vector<int> st;
    int ptr_idx = 0;
    void push(T val) {
        while (
            (int) st.size() > ptr_idx && vals[st.back()] >=
            val) {
            st.pop_back();
        }
        st.push_back((int) vals.size());
        vals.push_back(val);
    }
    void pop() {
        assert(ptr < (int) vals.size());
        if (ptr_idx < (int) st.size() && st[ptr_idx] == ptr) {
            ptr_idx++;
        }
        ptr++;
    }
    T get() {
        assert(ptr_idx < (int) st.size());
        return vals[st[ptr_idx]];
    }
    int front() {
        assert(!empty());
        return vals[ptr];
    }
    int back() {
        assert(!empty());
        return vals.back();
    }
    bool empty() { return (ptr == (int) vals.size()); }
    int size() { return ((int) vals.size() - ptr); }
};

```

2.6 Binary trie

Description: a binary trie that supports inserting, erasing and querying min/max $x \oplus val$ for a given x .

Time: $O(\text{BIT})$ per query

binary_trie.h, 53 lines

```

template<typename T, int BIT> struct BinaryTrie {
    struct Node {
        array<int, 2> next;
        int cnt;
        Node(): cnt(0) { next.fill(-1); }
    };
    vector<Node> trie;
    BinaryTrie() { trie.emplace_back(); }
    bool contains(T val) {
        for (int j = BIT - 1, i = 0; j >= 0; --j) {
            int x = (val >> j) & 1;
            int ni = trie[i].next[x];
            if (ni == -1 || trie[ni].cnt == 0) { return false; }
            i = ni;
        }
        return true;
    }
    void insert(T val) {
        for (int j = BIT - 1, i = 0; j >= 0; --j) {
            int x = (val >> j) & 1;
            if (trie[i].next[x] == -1) {
                trie[i].next[x] = trie.size();
                trie.emplace_back();
            }
            i = trie[i].next[x];
            ++trie[i].cnt;
        }
    }
    void erase(T val) {
        for (int j = BIT - 1, i = 0; j >= 0; --j) {
            int x = (val >> j) & 1;
            int ni = trie[i].next[x];
            assert(ni != -1 && trie[ni].cnt > 0);
            i = ni;
            --trie[i].cnt;
        }
    }
    T get_min(T val) {
        T ret = 0;
        for (int j = BIT - 1, i = 0; j >= 0; --j) {
            int x = (val >> j) & 1;
            int l = trie[i].next[x];
            int r = trie[i].next[x ^ 1];
            if (l == -1 || trie[l].cnt == 0) {
                i = r;
                ret |= static_cast<T>(1) << j;
            } else {
                i = l;
            }
        }
        return ret;
    }
};

```

2.7 Segment tree

Description: A segment tree with range updates and sum queries that supports three types of operations:

- Increase each value in range $[l, r]$ by x (i.e. $a[i] += x$).
- Set each value in range $[l, r]$ to x (i.e. $a[i] = x$).
- Determine the sum of values in range $[l, r]$.

segment_tree.h, 69 lines

```

struct SegmentTree {
    int n;
    vector<long long> tree, lazy_add, lazy_set;
    SegmentTree(int _n): n(_n) {

```

```

        int p = 1;
        while (p < n) { p *= 2; }
        tree.resize(p * 2);
        lazy_add.resize(p * 2);
        lazy_set.resize(p * 2);
    }
    long long merge(
        const long long& left, const long long& right) {
        return left + right;
    }
    void build(int id, int l, int r, const vector<int>& arr) {
        if (l == r) {
            tree[id] += arr[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(id * 2, l, mid, arr);
        build(id * 2 + 1, mid + 1, r, arr);
        tree[id] = merge(tree[id * 2], tree[id * 2 + 1]);
    }
    void push(int id, int l, int r) {
        if (lazy_set[id] == 0 && lazy_add[id] == 0) { return; }
        int mid = (l + r) >> 1;
        for (int child : {id * 2, id * 2 + 1}) {
            int range = (child == id * 2 ? mid - l + 1 : r - mid);
            if (lazy_set[id] != 0) {
                lazy_add[child] = 0;
                lazy_set[child] = lazy_set[id];
                tree[child] = range * lazy_set[id];
            }
            lazy_add[child] += lazy_add[id];
            tree[child] += range * lazy_add[id];
        }
        lazy_add[id] = lazy_set[id] = 0;
    }
    void update(int id, int l, int r, int u, int v,
        int amount, bool set_value = false) {
        if (r < u || l > v) { return; }
        if (u <= l && r <= v) {
            if (set_value) {
                tree[id] = 1LL * amount * (r - l + 1);
                lazy_set[id] = amount;
                lazy_add[id] = 0; // clear all previous updates.
            } else {
                tree[id] += 1LL * amount * (r - l + 1);
                lazy_add[id] += amount;
            }
            return;
        }
        push(id, l, r);
        int mid = (l + r) >> 1;
        update(id * 2, l, mid, u, v, amount, set_value);
        update(id * 2 + 1, mid + 1, r, u, v, amount, set_value);
        tree[id] = merge(tree[id * 2], tree[id * 2 + 1]);
    }
    long long get(int id, int l, int r, int u, int v) {
        if (r < u || l > v) { return 0; }
        if (u <= l && r <= v) { return tree[id]; }
        push(id, l, r);
        int mid = (l + r) >> 1;
        long long left = get(id * 2, l, mid, u, v);
        long long right = get(id * 2 + 1, mid + 1, r, u, v);
        return merge(left, right);
    }
};

```

2.8 Efficient segment tree

efficient_segment_tree.h, 33 lines

```
template<typename T> struct SegmentTree {
    int n;
    vector<T> tree;
    SegmentTree(int _n): n(_n), tree(2 * n) {}
    T merge(const T& left, const T& right) {
        return left + right;
    }
    template<typename G>
    void build(const vector<G>& initial) {
        assert((int) initial.size() == n);
        for (int i = 0; i < n; ++i) {
            tree[i + n] = initial[i];
        }
        for (int i = n - 1; i > 0; --i) {
            tree[i] = merge(tree[i * 2], tree[i * 2 + 1]);
        }
    }
    void modify(int i, int v) {
        tree[i + n] = v;
        for (i /= 2; i > 0; i /= 2) {
            tree[i] = merge(tree[i * 2], tree[i * 2 + 1]);
        }
    }
    T get_sum(int l, int r) {
        // sum of elements from l to r - 1.
        T ret{};
        for (l += n, r += n; l < r; l /= 2, r /= 2) {
            if (l & 1) { ret = merge(ret, tree[l++]); }
            if (r & 1) { ret = merge(ret, tree[--r]); }
        }
        return ret;
    }
};
```

2.9 Persistent lazy segment tree

persistent_lazy_segment_tree.h, 69 lines

```
struct Node {
    int lc, rc;
    long long val, lazy;
    bool has_changed = false;
    Node() {}
    Node(
        int _lc, int _rc, long long _val, long long _lazy =
        0):
        lc(_lc), rc(_rc), val(_val), lazy(_lazy) {}
};
struct PerSegmentTree {
    vector<Node> tree;
    int build(const vector<int>& arr, int l, int r) {
        if (l == r) {
            tree.emplace_back(-1, -1, arr[l]);
            return tree.size() - 1;
        }
        int mid = (l + r) / 2;
        int lc = build(arr, l, mid);
        int rc = build(arr, mid + 1, r);
        tree.emplace_back(lc, rc, tree[lc].val +
            tree[rc].val);
        return tree.size() - 1;
    }
    int add(int x, int l, int r, int u, int v, int amt) {
        if (l > v || r < u) { return x; }
        if (u <= l && r <= v) {
            tree.emplace_back(tree[x].lc, tree[x].rc,
                tree[x].val + 1LL * amt * (r - l + 1),
```

```
            tree[x].lazy + amt);
            tree.back().has_changed = true;
            return tree.size() - 1;
        }
        int mid = (l + r) >> 1;
        push(x, l, mid, r);
        int lc = add(tree[x].lc, l, mid, u, v, amt);
        int rc = add(tree[x].rc, mid + 1, r, u, v, amt);
        tree.emplace_back(
            lc, rc, tree[lc].val + tree[rc].val, 0);
        return tree.size() - 1;
    }
    long long get_sum(int x, int l, int r, int u, int v) {
        if (r < u || l > v) { return 0; }
        if (u <= l && r <= v) { return tree[x].val; }
        int mid = (l + r) / 2;
        push(x, l, mid, r);
        auto lhs = get_sum(tree[x].lc, l, mid, u, v);
        auto rhs = get_sum(tree[x].rc, mid + 1, r, u, v);
        return lhs + rhs;
    }
    void push(int x, int l, int mid, int r) {
        if (!tree[x].has_changed) { return; }
        Node left = tree[tree[x].lc];
        Node right = tree[tree[x].rc];
        tree.emplace_back(left);
        tree[x].lc = tree.size() - 1;
        tree.emplace_back(right);
        tree[x].rc = tree.size() - 1;

        tree[tree[x].lc].val += tree[x].lazy * (mid - l + 1);
        tree[tree[x].lc].lazy += tree[x].lazy;

        tree[tree[x].rc].val += tree[x].lazy * (r - mid);
        tree[tree[x].rc].lazy += tree[x].lazy;

        tree[tree[x].lc].has_changed = true;
        tree[tree[x].rc].has_changed = true;
        tree[x].lazy = 0;
        tree[x].has_changed = false;
    }
};
```

2.10 Lichao tree

Description: A segment tree that allows inserting a new line and querying for minimum value over all lines at point x.

Usage: useful in convex hull trick.

lichao_tree.h, 43 lines

```
const long long INF_LL = (long long) 4e18;

struct Line {
    long long a, b;
    Line(long long _a = 0, long long _b = INF_LL):
        a(_a), b(_b) {}
    long long operator()(long long x) { return a * x + b; }
};

struct SegmentTree { // min query
    int n;
    vector<Line> tree;
    SegmentTree() {}
    SegmentTree(int _n): n(1) {
        while (n < _n) { n *= 2; }
        tree.resize(n * 2);
    }
    void insert(int x, int l, int r, Line line) {
        if (l == r) {
            if (line(l) < tree[x](l)) { tree[x] = line; }
        }
```

```
        return;
    }
    int mid = (l + r) >> 1;
    bool b_left = line(l) < tree[x](l);
    bool b_mid = line(mid) < tree[x](mid);
    if (b_mid) { swap(tree[x], line); }
    if (b_left != b_mid) {
        insert(x * 2, l, mid, line);
    } else {
        insert(x * 2 + 1, mid + 1, r, line);
    }
}

long long query(int x, int l, int r, int at) {
    if (l == r) { return tree[x](at); }
    int mid = (l + r) >> 1;
    if (at <= mid) {
        return min(tree[x](at), query(x * 2, l, mid, at));
    } else {
        return min(
            tree[x](at), query(x * 2 + 1, mid + 1, r, at));
    }
}
};
```

2.11 Old driver tree (Chtholly tree)

Description: An optimized brute-force approach to deal with problems that have operation of setting an interval to the same number.

Note: only works when inputs are random, otherwise it will TLE.

old_driver_tree.h, 60 lines

```
#include "../number-theory/modmul.h"
struct ODT {
    map<int, long long> tree;
    using It = map<int, long long>::iterator;

    It split(int x) {
        It it = tree.upper_bound(x);
        assert(it != tree.begin());
        --it;
        if (it->first == x) { return it; }
        return tree.emplace(x, it->second).first;
    }

    void add(int l, int r, int amt) {
        It it_l = split(l);
        It it_r = split(r + 1);
        while (it_l != it_r) {
            it_l->second += amt;
            ++it_l;
        }
    }

    void set(int l, int r, int v) {
        It it_l = split(l);
        It it_r = split(r + 1);
        while (it_l != it_r) { tree.erase(it_l++); }
        tree[l] = v;
    }

    long long kth_smallest(int l, int r, int k) {
        // return the k-th smallest value in range [l..r]
        vector<pair<long long, int>>
            values; // pair(value, count)
        It it_l = split(l);
        It it_r = split(r + 1);
        while (it_l != it_r) {
            It prev = it_l++;
            values.emplace_back(
                prev->second, it_l->first - prev->first);
```

```

    }
    sort(values.begin(), values.end());
    for (auto [value, cnt] : values) {
        if (k <= cnt) { return value; }
        k -= cnt;
    }
    return -1;
}
int sum_of_xth_power(int l, int r, int x, int mod) {
    It it_l = split(l);
    It it_r = split(r + 1);
    int res = 0;
    while (it_l != it_r) {
        It prev = it_l++;
        res = (res + 1LL * (it_l->first - prev->first) *
                modpow(prev->second, x, mod)) %
            mod;
    }
    return res;
}
};

```

2.12 Disjoint sparse table

Description: range query on a static array.

Time: $O(1)$ per query.

disjoint_sparse_table.h, 37 lines

```

const int MOD = (int) 1e9 + 7;
struct DisjointSparseTable { // product queries.
    int n, h;
    vector<vector<int>> dst;
    vector<int> lg;
    DisjointSparseTable(int _n): n(_n) {
        h = 1; // in case n = 1: h = 0 !!.
        int p = 1;
        while (p < n) { p *= 2, h++; }
        lg.resize(p);
        lg[1] = 0;
        for (int i = 2; i < p; ++i) { lg[i] = 1 + lg[i / 2]; }
        dst.resize(h, vector<int>(n));
    }
    void build(const vector<int>& A) {
        for (int lv = 0; lv < h; ++lv) {
            int len = (1 << lv);
            for (int k = 0; k < n; k += len * 2) {
                int mid = min(k + len, n);
                dst[lv][mid - 1] = A[mid - 1] % MOD;
                for (int i = mid - 2; i >= k; --i) {
                    dst[lv][i] = 1LL * A[i] * dst[lv][i + 1] % MOD;
                }
                if (mid == n) { break; }
                dst[lv][mid] = A[mid] % MOD;
                for (int i = mid + 1; i < min(mid + len, n); ++i) {
                    dst[lv][i] = 1LL * A[i] * dst[lv][i - 1] % MOD;
                }
            }
        }
    }
    int get(int l, int r) {
        if (l == r) { return dst[0][l]; }
        int i = lg[l ^ r];
        return 1LL * dst[i][l] * dst[i][r] % MOD;
    }
};

```

2.13 Fenwick tree

Description: a minimal and simple data structure for point update and range

query

Note:

- For range update and point query, create a Fenwick tree on array D defined by $D_0 = A_0, D_i = A_i - A_{i-1}$
- For range update and range query, the idea is the same as above, but we can calculate the prefix as follow: $\sum_{i=0}^k A_i = \sum_{i=0}^k \sum_{j=0}^i D_j = (k+1) \sum_{i=0}^k D_i - \sum_{i=0}^k i D_i$, thus we can maintain two prefix sums, D_i and $i D_i$, with two Fenwick trees.

Time: $O(\log N)$

fenwick_tree.h, 32 lines

```

template<typename T> struct Fenwick {
    int n;
    vector<T> tree;
    Fenwick() {}
    Fenwick(int _n): n(_n), tree(n) {}
    void add(int i, T val) {
        while (i < n) {
            tree[i] += val;
            i |= (i + 1);
        }
    }
    T pref(int i) {
        T res{};
        while (i >= 0) {
            res += tree[i];
            i = (i & (i + 1)) - 1;
        }
        return res;
    }
    T query(int l, int r) { return pref(r) - pref(l - 1); }
    int lower_bound(T val) {
        int x = 0;
        T s{};
        for (int i = 1 << __lg(n); i > 0; i /= 2) {
            if (i + x - 1 < n && s + tree[x + i - 1] < val) {
                s += tree[x + i - 1];
                x += i;
            }
        }
        return x;
    }
};

```

2.14 Treap

Description: Treap is a type of self-balancing binary search tree. It is a combination of binary search tree and binary heap. The two main methods are split and merge. It is easy to implement and augment with additional information.

Time: $O(\log N)$.

treap.h, 95 lines

```

struct Node {
    int val, prior, cnt;
    bool rev;
    Node *left, *right;
    Node() {}
    Node(int _val):
        val(_val), prior(rng()), cnt(1), rev(false),
        left(nullptr), right(nullptr) {}
    friend int get_cnt(Node *n) { return n ? n->cnt : 0; }
    void pull() { cnt = get_cnt(left) + 1 + get_cnt(right); }
    void push() {
        if (!rev) { return; }
        rev = false;
        swap(left, right);
        if (left) { left->rev ^= 1; }
    }
};

```

```

    if (right) { right->rev ^= 1; }
};
struct Treap {
    Node *root;
    bool implicit_key;
    Treap(bool _implicit_key = true):
        root(nullptr), implicit_key(_implicit_key) {}
    bool go_right(Node *treap, int pos_or_val) {
        if (implicit_key) {
            int local_idx = get_cnt(treap->left);
            return local_idx <= pos_or_val;
        }
        return treap->val <= pos_or_val;
    }
    pair<Node *, Node *> split(Node *treap, int pos_or_val)
    {
        // normal treap -> Left: all nodes having val <= val
        // implicit treap -> Left: all nodes having index <= pos
        if (!treap) { return {}; }
        treap->push();
        if (go_right(treap, pos_or_val)) {
            if (implicit_key) {
                pos_or_val -= (get_cnt(treap->left) + 1);
            }
            auto pr = split(treap->right, pos_or_val);
            treap->right = pr.first;
            treap->pull();
            return {treap, pr.second};
        } else {
            auto pl = split(treap->left, pos_or_val);
            treap->left = pl.second;
            treap->pull();
            return {pl.first, treap};
        }
    }
    tuple<Node *, Node *, Node *> split(int u, int v) {
        auto [l, rem] = split(root, u - 1);
        auto [mid, r] = split(rem, v - (implicit_key ? u : 0));
        return {l, mid, r};
    }
    Node *merge(Node *l, Node *r) {
        if (!l || !r) { return (l ? l : r); }
        if (l->prior < r->prior) {
            l->push();
            l->right = merge(l->right, r);
            l->pull();
            return l;
        } else {
            r->push();
            r->left = merge(l, r->left);
            r->pull();
            return r;
        }
    }
    void insert(int pos, int val) {
        auto [l, r] = split(root, pos - 1);
        root = merge(merge(l, new Node(val)), r);
    }
    void insert(int val) { insert(val, val); }
    void erase(int u, int v) {
        auto [l, mid, r] = split(u, v);
        root = merge(l, r);
    }
    void reverse(int u, int v) {
        auto [l, mid, r] = split(u, v);
        mid->rev ^= true;
    }
};

```



```

    root = merge(merge(l, mid), r);
}
int get_kth(Node *treap, int k) {
    if (!treap) { return (int) 1e9; }
    treap->push();
    int sz = get_cnt(treap->left) + 1;
    if (sz == k) {
        return treap->val;
    } else if (sz < k) {
        return get_kth(treap->right, k - sz);
    }
    return get_kth(treap->left, k);
}
};

```

2.15 Splay tree

Description: a type of self-balancing binary search tree, when a node is accessed, a splay operation is performed on that node to make it become the root of the tree.

Time: amortized time complexity is $O(\log N)$.

splay_tree.h, 135 lines

```

struct Node {
    int val, cnt;
    bool rev;
    Node *left, *right, *par;
    Node() {}
    Node(int _val = 0):
        val(_val), cnt(1), rev(false), left(nullptr),
        right(nullptr), par(nullptr) {}
    friend int get_cnt(Node *n) { return n ? n->cnt : 0; }
    void pull() {
        cnt = get_cnt(left) + 1 + get_cnt(right);
        if (left) { left->par = this; }
        if (right) { right->par = this; }
    }
    void push() {
        if (!rev) { return; }
        rev = false;
        swap(left, right);
        if (left) { left->rev ^= 1; }
        if (right) { right->rev ^= 1; }
    }
};
bool is_root(Node *n) {
    return (n != nullptr && n->par == nullptr);
}
struct SplayTree {
    void splay(Node *u) {
        if (u == nullptr) { return; }
        u->push();
        while (!is_root(u)) {
            Node *par = u->par;
            if (!is_root(par)) {
                if ((par->left == u) == (par->par->left == par)) {
                    // zig-zig
                    rotate(par);
                } else {
                    // zig-zag
                    rotate(u);
                }
            }
            rotate(u);
        }
        u->pull();
    }
    Node *merge(Node *u, Node *v) {
        if (!u) { return v; }

```

```

        if (!v) { return u; }
        while (true) {
            u->push();
            Node *next = u->right;
            if (next == nullptr) { break; }
            u = next;
        }
        splay(u);
        splay(v);
        assert(u->right == nullptr);
        u->right = v;
        u->pull();
        return u;
    }
    void rotate(Node *u) {
        Node *par = u->par;
        assert(par != nullptr);
        par->push();
        u->push();
        u->par = par->par;
        if (par->par != nullptr) {
            if (u->par->left == par) {
                u->par->left = u;
            } else {
                u->par->right = u;
            }
        }
        if (par->left == u) {
            par->left = u->right;
            u->right = par;
        } else {
            par->right = u->left;
            u->left = par;
        }
        par->pull();
        u->pull();
    }
    Node *node_at_index(Node *n, int pos) {
        if (pos < 0 || pos >= get_cnt(n)) { return nullptr; }
        n->push();
        int idx = get_cnt(n->left);
        if (idx == pos) {
            return n;
        } else if (idx < pos) {
            return node_at_index(n->right, pos - idx - 1);
        } else {
            return node_at_index(n->left, pos);
        }
    }
    pair<Node *, Node *> split(Node *n, int pos) {
        if (pos < 0) { return {nullptr, n}; }
        if (pos >= get_cnt(n) - 1) { return {n, nullptr}; }
        Node *l = node_at_index(n, pos);
        splay(l);
        Node *r = l->right;
        l->right = nullptr;
        r->par = nullptr;
        l->pull();
        return {l, r};
    }
    tuple<Node *, Node *, Node *> split(
        Node *n, int u, int v) {
        auto [l, rem] = split(n, u - 1);
        auto [mid, r] = split(rem, v - u);
        return {l, mid, r};
    }
    Node *reverse(Node *n, int u, int v) {
        auto [l, mid, r] = split(n, u, v);
        mid->rev ^= 1;

```

```

        Node *ret = merge(l, merge(mid, r));
        return ret;
    }
    Node *insert(Node *n, int pos, int val) {
        auto [l, r] = split(n, pos - 1);
        return merge(l, merge(new Node(val), r));
    }
    Node *erase(Node *n) {
        if (!n) { return nullptr; }
        splay(n);
        Node *left = n->left, *right = n->right;
        n->left = n->right = nullptr;
        if (left) { left->par = nullptr; }
        if (right) { right->par = nullptr; }

        Node *ret = merge(left, right);
        if (ret != nullptr) { ret->par = n->par; }
        return ret;
    }
};

```

2.16 Line container

Description: container that allow you can add lines in form $ax + b$ and query maximum value at x .

line_container.h, 49 lines

```

using num_t = int;
struct Line {
    num_t a, b; // ax + b
    mutable num_t
        x; // x-intersect with the next line in the hull
    bool operator<(const Line& other) const {
        return a < other.a;
    }
    bool operator<(num_t other_x) const {
        return x < other_x;
    }
};
struct LineContainer: multiset<Line, less<>> { //
    max-query
    // for doubles, use INF = 1 / 0.0
    static const num_t INF = numeric_limits<num_t>::max();

    num_t floor_div(num_t a, num_t b) {
        return a / b - ((a ^ b) < 0 && a % b != 0);
    }
    bool isect(iterator u, iterator v) {
        if (v == end()) {
            u->x = INF;
            return false;
        }
        if (u->a == v->a) {
            u->x = (u->b > v->b ? INF : -INF);
        } else {
            u->x = floor_div(v->b - u->b, u->a - v->a);
        }
        return u->x >= v->x;
    }
    void add(num_t a, num_t b) {
        auto z = insert({a, b, 0});
        while (isect(y, z)) { z = erase(z); }
        if (x != begin() && isect(--x, y)) {
            y = erase(y);
            isect(x, y);
        }
        while ((y = x) != begin() && (--x)->x >= y->x) {
            isect(x, erase(y));
        }
    }
};

```

```

num_t query(num_t x) {
    assert(!empty());
    auto it = *lower_bound(x);
    return it.a * x + it.b;
}
};

```

2.17 Wavelet matrix

Description: an efficient, fast, and lightweight data structure supporting queries like k-th smallest element in range or count lowers value in range.

Time: $O(\log(\max\{A_i\}))$

wavelet_matrix.h, 115 lines

```

struct bit_vector {
    static constexpr int word_size =
        numeric_limits<uint64_t>::digits;
    vector<uint64_t> block;
    vector<uint32_t> pref; // pref is 1-indexed
    bit_vector() {}
    bit_vector(const vector<bool>& a) {
        int n = (int) a.size();
        block.resize(n / word_size + 1);
        pref.resize(n / word_size + 1);
        for (int i = 0; i < n; ++i) {
            block[i / word_size] |= static_cast<uint64_t>(a[i])
                << (i % word_size);
        }
        for (int i = 0; i < (int) block.size() - 1; ++i) {
            pref[i + 1] =
                pref[i] + __builtin_popcountll(block[i]);
        }
    }
    uint32_t rank0(uint32_t i) const { return i - rank1(i); }
    uint32_t rank1(uint32_t i) const {
        return pref[i / word_size] +
            __builtin_popcountll(block[i / word_size] &
                ~static_cast<uint64_t>(0) << (i % word_size));
    }
};

template<typename key_type> struct WaveletMatrix {
    int n, max_level;
    vector<bit_vector> mat;
    WaveletMatrix() {}
    WaveletMatrix(vector<key_type> a): n(a.size()) {
        key_type max_v = *max_element(a.begin(), a.end());
        max_level = __lg(max<key_type>(max_v, 1)) + 1;
        mat.resize(max_level);
        for (int h = max_level - 1; h >= 0; --h) {
            vector<bool> b(n);
            for (int i = 0; i < n; ++i) { b[i] = test(a[i], h); }
            mat[h] = bit_vector(b);
            vector<key_type> v0, v1;
            for (int i = 0; i < n; ++i) {
                if (test(a[i], h)) {
                    v1.emplace_back(a[i]);
                } else {
                    v0.emplace_back(a[i]);
                }
            }
            const auto iter =
                copy(v0.cbegin(), v0.cend(), a.begin());
            copy(v1.begin(), v1.end(), iter);
        }
    }
};

```

```

static bool test(key_type mask, int i) {
    return (mask >> i) & static_cast<key_type>(1);
}

static void set(key_type& mask, int i) {
    mask |= static_cast<key_type>(1) << i;
}

key_type kth(int first, int last, int k) const {
    // return the k-th (0-indexed) smallest element in
    // range
    // [first, last)
    assert(0 <= first && first < last && last <= n);
    assert(k < last - first);
    key_type ret = 0;
    for (int h = max_level - 1; h >= 0; --h) {
        const bit_vector& v = mat[h];
        int cnt0 = v.rank0(last) - v.rank0(first);
        if (k < cnt0) {
            first = v.rank0(first);
            last = v.rank0(last);
        } else {
            set(ret, h);
            k -= cnt0;
            int zeros = v.rank0(n);
            first = zeros + v.rank1(first);
            last = zeros + v.rank1(last);
        }
    }
    return ret;
}

key_type count_lower(
    int first, int last, key_type val) const {
    // count first <= i < last s.t. a[i] < val
    assert(0 <= first && first < last && last <= n);
    if (val >= static_cast<key_type>(1) << max_level) {
        return last - first;
    }
    key_type ret = 0;
    for (int h = max_level - 1; h >= 0; --h) {
        const bit_vector& v = mat[h];
        if (!test(val, h)) {
            first = v.rank0(first);
            last = v.rank0(last);
        } else {
            ret += v.rank0(last) - v.rank0(first);
            int zeros = v.rank0(n);
            first = zeros + v.rank1(first);
            last = zeros + v.rank1(last);
        }
    }
    return ret;
}

key_type count_upper(
    int first, int last, key_type val) const {
    // count first <= i < last s.t. a[i] >= val
    return last - first - count_lower(first, last, val);
}

key_type range_count(
    int first, int last, key_type A, key_type B) const {
    // count first <= i < last s.t. A <= a[i] < B
    return count_lower(first, last, B) -
        count_lower(first, last, A);
}
};

```

3 Mathematics

3.1 Trigonometry

3.1.1 Sum - difference identities

$$\sin(u \pm v) = \sin(u) \cos(v) \pm \cos(u) \sin(v)$$

$$\cos(u \pm v) = \cos(u) \cos(v) \mp \sin(u) \sin(v)$$

$$\tan(u \pm v) = \frac{\tan(u) \pm \tan(v)}{1 \mp \tan(u) \tan(v)}$$

3.1.2 Sum to product identities

$$\cos(u) + \cos(v) = 2 \cos\left(\frac{u+v}{2}\right) \cos\left(\frac{u-v}{2}\right)$$

$$\cos(u) - \cos(v) = -2 \sin\left(\frac{u+v}{2}\right) \sin\left(\frac{u-v}{2}\right)$$

$$\sin(u) + \sin(v) = 2 \sin\left(\frac{u+v}{2}\right) \cos\left(\frac{u-v}{2}\right)$$

$$\sin(u) - \sin(v) = 2 \cos\left(\frac{u+v}{2}\right) \sin\left(\frac{u-v}{2}\right)$$

3.1.3 Product identities

$$\cos(u) \cos(v) = \frac{1}{2} [\cos(u+v) + \cos(u-v)]$$

$$\sin(u) \sin(v) = -\frac{1}{2} [\cos(u+v) - \cos(u-v)]$$

$$\sin(u) \cos(v) = \frac{1}{2} [\sin(u+v) + \sin(u-v)]$$

3.1.4 Double - triple angle identities

$$\sin(2u) = 2 \sin(u) \cos(u)$$

$$\cos(2u) = 2 \cos^2(u) - 1 = 1 - 2 \sin^2(u)$$

$$\tan(2u) = \frac{2 \tan(u)}{1 - \tan^2(u)}$$

$$\sin(3u) = 3 \sin(u) - 4 \sin^3(u)$$

$$\cos(3u) = 4 \cos^3(u) - 3 \cos(u)$$

$$\tan(3u) = \frac{3 \tan(u) - \tan^3(u)}{1 - 3 \tan^2(u)}$$

3.2 Sums

$$\sum_{i=0}^n ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}, \quad c \neq 1$$

$$\sum_{i=a}^b c^i = \frac{c^{b+1} - c^a}{c-1}, \quad c \neq 1$$

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

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

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

$$\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

$$\sum_{i=1}^n i^5 = \frac{n^2(n+1)^2(2n^2+2n-1)}{12}$$

$$\sum_{i=1}^n i^6 = \frac{n(n+1)(2n+1)(3n^4+6n^3-3n+1)}{42}$$

$$\sum_{i=1}^n i^7 = \frac{n^2(n+1)^2(3n^4+6n^3-n^2-4n+2)}{24}$$

$$\sum_{i=0}^n \binom{n}{i} a^{n-i} b^i = (a+b)^n$$

$$\sum_{i=0}^n i \binom{n}{i} = n2^{n-1}$$

$$\sum_{i=0}^n \frac{\binom{n}{i}}{i+1} = \frac{2^{n+1}-1}{n+1}$$

$$\sum_{k=0}^m \binom{n+k}{n} = \binom{n+m+1}{n+1}$$

$$\sum_{i=k}^n \binom{i}{k} = \binom{n+1}{k+1}$$

3.3 Pythagorean triple

- A Pythagorean triple is a triple of positive integers a , b , and c such that $a^2 + b^2 = c^2$.
- If (a, b, c) is a Pythagorean triple, then so is (ka, kb, kc) for any positive integer k .
- A primitive Pythagorean triple is one in which a , b , and c are coprime.

• Generating Pythagorean triple

- Euclid's formula: with arbitrary $0 < n < m$, then:

$$a = m^2 - n^2, \quad b = 2mn, \quad c = m^2 + n^2$$

form a Pythagorean triple.

- To obtain primitive Pythagorean triple, this condition must hold: m and n are coprime, m and n have opposite parity.

4 String

4.1 Prefix function

Description: the prefix function of a string s is defined as an array pi of length n , where $pi[i]$ is the length of the longest proper prefix of the sub-string $s[0..i]$ which is also a suffix of this sub-string.

Time: $O(|S|)$.

prefix_function.h, 11 lines

```
vector<int> prefix_function(const string& s) {
    int n = (int) s.length();
    vector<int> pi(n);
    pi[0] = 0;
    for (int i = 1; i < n; ++i) {
        int j = pi[i - 1]; // try length pi[i - 1] + 1.
        while (j > 0 && s[j] != s[i]) { j = pi[j - 1]; }
        if (s[j] == s[i]) { pi[i] = j + 1; }
    }
    return pi;
}
```

4.2 Z function

Description: for a given string ' s ', $z[i]$ = longest common prefix of ' s ' and suffix starting at ' i '. $z[0]$ is generally not well defined (this implementation below assume $z[0] = 0$).

Time: $O(n)$.

z_function.h, 17 lines

```
vector<int> z_function(const string& s) {
    int n = (int) s.size();
    vector<int> z(n);
    z[0] = 0;
    // [l, r)
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i < r) { z[i] = min(r - i, z[i - l]); }
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            ++z[i];
        }
        if (i + z[i] > r) {
            l = i;
            r = i + z[i];
        }
    }
    return z;
}
```

4.3 Counting occurrences of each prefix

Description: count the number of occurrences of each prefix in the given string.

Time: $O(n)$.

counting_occur_of_prefix.h, 14 lines

```
#include "prefix_function.h"
```

```
vector<int> count_occurrences(const string& s) {
    vector<int> pi = prefix_function(s);
    int n = (int) s.size();
    vector<int> ans(n + 1);
    for (int i = 0; i < n; ++i) { ans[pi[i]]++; }
    for (int i = n - 1; i > 0; --i) {
        ans[pi[i - 1]] += ans[i];
    }
    for (int i = 0; i <= n; ++i) { ans[i]++; }
    return ans;
    // Input: ABACABA
    // Output: 4 2 2 1 1 1 1
}
```

4.4 Knuth–Morris–Pratt algorithm

Description: searching for a sub-string in a string.

Time: $O(N + M)$.

KMP.h, 11 lines

```
#include "prefix_function.h"
vector<int> KMP(const string& text, const string&
    pattern) {
    int n = (int) pattern.length();
    string combined = pattern + '$' + text;
    vector<int> pi = prefix_function(combined);
    vector<int> indices;
    for (int i = 0; i < (int) combined.length(); ++i) {
        if (pi[i] == n) { indices.push_back(i - 2 * n); }
    }
    return indices;
}
```

4.5 Suffix array

Description: suffix array is a sorted array of all the suffixes of a given string.

Usage:

- $sa[i]$ = starting index of the i -th smallest suffix.
- $rank[i]$ = rank of the suffix starting at ' i '.
- $lcp[i]$ = longest common prefix between ' $sa[i - 1]$ ' and ' $sa[i]$ '
- for arbitrary ' u v ', let $i = rank[u] - 1$, $j = rank[v] - 1$ (assume $i < j$), then $longest_common_prefix(u, v) = \min(lcp[i + 1], lcp[i + 2], \dots, lcp[j])$

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

suffix_array.h, 55 lines

```
struct SuffixArray {
    string s;
    int n, lim;
    vector<int> sa, lcp, rank;
    SuffixArray(const string& _s, int _lim = 256):
        s(_s), n(s.length() + 1), lim(_lim), sa(n), lcp(n),
        rank(n) {
        s += '$';
        build();
        kasai();
        sa.erase(sa.begin());
        lcp.erase(lcp.begin());
        rank.pop_back();
        s.pop_back();
    }
    void build() {
        vector<int> nrank(n), norder(n), cnt(max(n, lim));
        for (int i = 0; i < n; ++i) {
            sa[i] = i;
            rank[i] = s[i];
        }
        for (int k = 0, rank_cnt = 0; rank_cnt < n - 1;
            k = max(1, k * 2), lim = rank_cnt + 1) {
            for (int i = 0; i < n; ++i) {
                norder[i] = (sa[i] - k + n) % n;
            }
        }
    }
};
```

```

    cnt[rank[i]]++;
}
for (int i = 1; i < lim; ++i) {
    cnt[i] += cnt[i - 1];
}
for (int i = n - 1; i >= 0; --i) {
    sa[--cnt[rank[norder[i]]]] = norder[i];
}
rank[sa[0]] = rank_cnt = 0;
for (int i = 1; i < n; ++i) {
    int u = sa[i], v = sa[i - 1];
    int nu = (u + k) % n, nv = (v + k) % n;
    if (rank[u] != rank[v] || rank[nu] != rank[nv]) {
        ++rank_cnt;
    }
    nrank[sa[i]] = rank_cnt;
}
for (int i = 0; i < rank_cnt + 1; ++i) { cnt[i] = 0; }
rank.swap(nrank);
}
}

void kasai() {
    for (int i = 0, k = 0; i < n - 1; ++i, k = max(0, k - 1)) {
        int j = sa[rank[i] - 1];
        while (s[i + k] == s[j + k]) { k++; }
        lcp[rank[i]] = k;
    }
}
};

```

4.6 Suffix array slow

Description: an easier and shorter implementation of suffix array but run a bit slower.

Time: $O(N \log^2 N)$.

suffix_array_slow.h, 43 lines

```

struct SuffixArraySlow {
    string s;
    int n;
    vector<int> sa, lcp, rank;
    SuffixArraySlow(const string& _s):
        s(_s), n((int) s.size() + 1), sa(n), lcp(n), rank(n) {
        s += '$';
        build();
        kasai();
        sa.erase(sa.begin());
        lcp.erase(lcp.begin());
        rank.pop_back();
        s.pop_back();
    }
    bool comp(int i, int j, int k) {
        return make_pair(rank[i], rank[(i + k) % n]) <
            make_pair(rank[j], rank[(j + k) % n]);
    }
    void build() {
        vector<int> nrank(n);
        for (int i = 0; i < n; ++i) {
            sa[i] = i;
            rank[i] = s[i];
        }
        for (int k = 0; k < n; k = max(1, k * 2)) {
            stable_sort(sa.begin(), sa.end(),
                [&](int i, int j) { return comp(i, j, k); });
            for (int i = 0, cnt = 0; i < n; ++i) {
                if (i > 0 && comp(sa[i - 1], sa[i], k)) { ++cnt; }
                nrank[sa[i]] = cnt;
            }
        }
    }
};

```

```

    }
    rank.swap(nrank);
}
}

void kasai() {
    for (int i = 0, k = 0; i < n - 1; ++i, k = max(0, k - 1)) {
        int j = sa[rank[i] - 1];
        while (s[i + k] == s[j + k]) { ++k; }
        lcp[rank[i]] = k;
    }
}
};

```

4.7 Manacher's algorithm

Description: for each position, computes $d[0][i]$ = half length of longest palindrome centered on i (rounded up), $d[1][i]$ = half length of longest palindrome centered on i and $i - 1$.

Time: $O(N)$.

manacher.h, 23 lines

```

array<vector<int>, 2> manacher(const string& s) {
    int n = (int) s.size();
    array<vector<int>, 2> d;
    for (int z = 0; z < 2; ++z) {
        d[z].resize(n);
        int l = 0, r = 0;
        for (int i = 0; i < n; ++i) {
            int mirror = l + r - i + z;
            d[z][i] = (i < r ? min(d[z][mirror], r - i) : 0);
            int L = i - d[z][i] - z, R = i + d[z][i];
            while (L >= 0 && R < n && s[L] == s[R]) {
                d[z][i]++;
                L--;
                R++;
            }
            if (R > r) {
                l = L;
                r = R;
            }
        }
        return d;
    }
}

```

4.8 Trie

Description: a rooted tree in which each edge is labeled with a character.

Usage:

- Check if a string exists in the set of strings.

Time: $O(N)$ for each operation where N is the length of the string.

trie.h, 36 lines

```

struct Trie {
    const static int ALPHABET = 26;
    const static char minChar = 'a';
    struct Vertex {
        int next[ALPHABET];
        bool leaf;
        Vertex() {
            leaf = false;
            fill(next, next + ALPHABET, -1);
        }
    };
    vector<Vertex> trie;
    Trie() { trie.emplace_back(); }

    void insert(const string& s) {
        int i = 0;
    }
};

```

```

    for (const char& ch : s) {
        int j = ch - minChar;
        if (trie[i].next[j] == -1) {
            trie[i].next[j] = trie.size();
            trie.emplace_back();
        }
        i = trie[i].next[j];
    }
    trie[i].leaf = true;
}

bool find(const string& s) {
    int i = 0;
    for (const char& ch : s) {
        int j = ch - minChar;
        if (trie[i].next[j] == -1) { return false; }
        i = trie[i].next[j];
    }
    return (trie[i].leaf ? true : false);
}
};

```

4.9 Aho-Corasick

aho_corasick.h, 71 lines

```

struct Vertex {
    int next[30];
    int output;
    int link;
    int ed, st;

    Vertex() {
        fill(begin(next), end(next), 0);
        output = link = 0;
        ed = st = 0;
    }
};

const int maxn = 1e5 + 3;
vector<Vertex> trie(1);
vector<int> adj[maxn];
int back[maxn], pre[maxn], val[maxn], vs[maxn];
map<string, int> mp;
string str[maxn];
int n;

void add_edge(string s, int idx) {
    int u = 0;

    for (auto j : s) {
        if (trie[u].next[j - 'a'] == 0) {
            trie[u].next[j - 'a'] = trie.size();
            trie.emplace_back();
        }
        u = trie[u].next[j - 'a'];
    }
    trie[u].output++;
    if (trie[u].st == 0) { trie[u].st = idx; }

    back[idx] = trie[u].ed;
    trie[u].ed = idx;
}

void get_link() {
    int u = 0;
    queue<int> q;

    for (int i = 0; i < 26; i++) {
        if (trie[u].next[i] != 0) { q.push(trie[u].next[i]); }
    }
}

```

```

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

    for (int i = 0; i < 26; i++) {
        int v = trie[u].next[i];
        if (v != 0) {
            trie[v].link = trie[trie[u].link].next[i];
            int y = trie[v].link;
            if (trie[v].ed == 0) {
                trie[v].ed = trie[y].ed;
            } else {
                back[trie[v].ed] = trie[y].ed;
            }
            q.push(v);
        } else {
            trie[u].next[i] = trie[trie[u].link].next[i];
        }
    }
}

for (int i = 1; i <= n; i++) {
    adj[back[i]].push_back(i);
}
}

```

4.10 Hashing

hash61.h, 63 lines

```

struct Hash61 {
    static const uint64_t MOD = (1LL << 61) - 1;
    static uint64_t BASE;
    static vector<uint64_t> pw;
    uint64_t addmod(uint64_t a, uint64_t b) const {
        a += b;
        if (a >= MOD) { a -= MOD; }
        return a;
    }
    uint64_t submod(uint64_t a, uint64_t b) const {
        a += MOD - b;
        if (a >= MOD) { a -= MOD; }
        return a;
    }
    uint64_t mulmod(uint64_t a, uint64_t b) const {
        uint64_t low1 = (uint32_t) a, high1 = (a >> 32);
        uint64_t low2 = (uint32_t) b, high2 = (b >> 32);

        uint64_t low = low1 * low2;
        uint64_t mid = low1 * high2 + low2 * high1;
        uint64_t high = high1 * high2;

        uint64_t ret = (low & MOD) + (low >> 61) + (high << 3) +
            (mid >> 29) + (mid << 35 >> 3) + 1;
        // ret %= MOD;
        ret = (ret >> 61) + (ret & MOD);
        ret = (ret >> 61) + (ret & MOD);
        return ret - 1;
    }
}

void ensure_pw(int m) {
    int sz = (int) pw.size();
    if (sz >= m) { return; }
    pw.resize(m);
    for (int i = sz; i < m; ++i) {
        pw[i] = mulmod(pw[i - 1], BASE);
    }
}

vector<uint64_t> pref;
int n;

```

```

template<typename T>
Hash61(const T& s) { // strings or arrays.
    n = (int) s.size();
    ensure_pw(n);
    pref.resize(n + 1);
    pref[0] = 0;
    for (int i = 0; i < n; ++i) {
        pref[i + 1] = addmod(mulmod(pref[i], BASE), s[i]);
    }
}

inline uint64_t operator()(
    const int from, const int to) const {
    assert(0 <= from && from <= to && to < n);
    // pref[to + 1] - pref[from] * pw[to - from + 1]
    return submod(
        pref[to + 1], mulmod(pref[from], pw[to - from + 1]));
}
};

mt19937 rnd((unsigned int) chrono::steady_clock::now()
    .time_since_epoch()
    .count());
uint64_t Hash61::BASE = (MOD >> 2) + rnd() % (MOD >> 1);
vector<uint64_t> Hash61::pw = vector<uint64_t>(1, 1);

```

4.11 Minimum rotation

Description: finds the lexicographically smallest rotation of a string.

Usage: rotate(v.begin(), v.begin() + minRotation(v), v.end())

Time: $O(N)$.

min_rotation.h, 19 lines

```

#pragma once

int minRotation(string s) {
    int a = 0, n = (int) s.size();
    s += s;
    for (int b = 0; b < n; ++b) {
        for (int k = 0; k < n; ++k) {
            if (a + k == b || s[a + k] < s[b + k]) {
                b += max(0, k - 1);
                break;
            }
            if (s[a + k] > s[b + k]) {
                a = b;
                break;
            }
        }
    }
    return a;
}

```

5 Numerical

5.1 Fast Fourier transform

Description: a fast algorithm for multiplying two polynomials.

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

fast_fourier_transform.h, 51 lines

```

const double PI = acos(-1);
using Comp = complex<double>;
int reverse_bit(int n, int lg) {
    int ret = 0;
    for (int i = 0; i < lg; ++i) {
        if ((n >> i) & 1) { ret |= (1 << (lg - i - 1)); }
    }
    return ret;
}

```

```

void fft(vector<Comp>& a, bool invert = false) {
    int n = (int) a.size();
    int lg = 0;
    while ((1 << lg) < n) { ++lg; }
    for (int i = 0; i < n; ++i) {
        int rev_i = reverse_bit(i, lg);
        if (i < rev_i) { swap(a[i], a[rev_i]); }
    }
    for (int len = 2; len <= n; len *= 2) {
        double angle = 2 * PI / len * (invert ? -1 : 1);
        Comp w_base(cos(angle), sin(angle));
        for (int i = 0; i < n; i += len) {
            Comp w(1);
            for (int j = i; j < i + len / 2; ++j) {
                Comp u = a[j], v = a[j + len / 2];
                a[j] = u + w * v;
                a[j + len / 2] = u - w * v;
                w *= w_base;
            }
        }
    }
    if (invert) {
        for (int i = 0; i < n; ++i) { a[i] /= n; }
    }
}

template<typename T, typename G>
vector<int64_t> mult(
    const vector<T>& a, const vector<G>& b) {
    vector<Comp> A(a.begin(), a.end()), B(b.begin(),
        b.end());
    int n = a.size(), m = b.size(), p = 1;
    while (p < n + m) { p *= 2; }
    A.resize(p), B.resize(p);
    fft(A, false);
    fft(B, false);
    for (int i = 0; i < p; ++i) { A[i] *= B[i]; }
    fft(A, true);
    vector<int64_t> res(n + m - 1);
    for (int i = 0; i < n + m - 1; ++i) {
        res[i] = (int64_t) round(A[i].real());
    }
    return res;
}

```

6 Number Theory

6.1 Euler's totient function

- Euler's totient function, also known as **phi-function** $\phi(n)$ counts the number of integers between 1 and n inclusive, that are **coprime to** n .
- Properties:
 - Divisor sum property: $\sum_{d|n} \phi(d) = n$.
 - $\phi(n)$ is a **prime number** when $n = 3, 4, 6$.
 - If p is a prime number, then $\phi(p) = p - 1$.
 - If p is a prime number and $k \geq 1$, then $\phi(p^k) = p^k - p^{k-1}$.
 - If a and b are **coprime**, then $\phi(ab) = \phi(a) \cdot \phi(b)$.

- In general, for **not coprime** a and b , with $d = \gcd(a, b)$ this equation holds: $\phi(ab) = \phi(a) \cdot \phi(b) \cdot \frac{d}{\phi(d)}$.
- With $n = p_1^{k_1} \cdot p_2^{k_2} \cdots p_m^{k_m}$:

$$\begin{aligned}\phi(n) &= \phi(p_1^{k_1}) \cdot \phi(p_2^{k_2}) \cdots \phi(p_m^{k_m}) \\ &= n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_m}\right)\end{aligned}$$

- Application in Euler's theorem:
 - If $\gcd(a, M) = 1$ (i.e. a and M are coprime), then:

$$a^{\phi(M)} \equiv 1 \pmod{M} \Rightarrow \begin{cases} a^n \equiv a^{n \bmod \phi(M)} \pmod{M} \\ a^{\phi(M)-1} \equiv a^{-1} \pmod{M} \end{cases}$$

- In general, for arbitrary a , M and $n \geq \log_2 M$:

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

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

phi_euler_totient_function.h, 12 lines

```
const int MAXN = (int) 2e5;
int etf[MAXN + 1];
void sieve(int n) {
    for (int i = 0; i <= n; ++i) { etf[i] = i; }
    for (int i = 2; i <= n; ++i) {
        if (etf[i] == i) {
            for (int j = i; j <= n; j += i) {
                etf[j] -= etf[j] / i;
            }
        }
    }
}
```

6.2 Mobius function

- For a positive integer $n = p_1^{k_1} \cdot p_2^{k_2} \cdots p_m^{k_m}$:

$$\mu(n) = \begin{cases} 1, & \text{if } n = 1 \\ 0, & \text{if } \exists k_i > 1 \\ (-1)^m & \text{otherwise} \end{cases}$$

- Properties:
 - $\sum_{d|n} \mu(d) = [n = 1]$.
 - If a and b are **coprime**, then $\mu(ab) = \mu(a) \cdot \mu(b)$.

- Mobius inversion: let f and g be arithmetic functions:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) g(d)$$

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

mobius_function.h, 8 lines

```
const int MAXN = (int) 2e5;
int mu[MAXN + 1];
void sieve(int n) {
    mu[1] = 1;
    for (int i = 1; i <= n; ++i) {
        for (int j = 2 * i; j <= n; j += i) { mu[j] -= mu[i]; }
    }
}
```

6.3 Primes

Approximating the number of primes up to n :

n	$\pi(n)$	$\frac{n}{\ln n - 1}$
100 ($1e^2$)	25	28
500 ($5e^2$)	95	96
1000 ($1e^3$)	168	169
5000 ($5e^3$)	669	665
10000 ($1e^4$)	1229	1218
50000 ($5e^4$)	5133	5092
100000 ($1e^5$)	9592	9512
500000 ($5e^5$)	41538	41246
1000000 ($1e^6$)	78498	78030
5000000 ($5e^6$)	348513	346622

($\pi(n)$ = the number of primes less than or equal to n , $\frac{n}{\ln n - 1}$ is used to approximate $\pi(n)$).

6.4 Wilson's theorem

A positive integer n is a prime if and only if:

$$(n - 1)! \equiv n - 1 \pmod{n}$$

6.5 Zeckendorf's theorem

The Zeckendorf's theorem states that every positive integer n can be represented uniquely as a sum of one or more distinct non-consecutive Fibonacci numbers. For example:

$$64 = 55 + 8 + 1$$

$$85 = 55 + 21 + 8 + 1$$

6.6 Chicken McNugget theorem

The Chicken McNugget theorem states that for any two relatively prime positive integers n, m , the greatest integer that

cannot be written in the form $a \cdot n + b \cdot m$ for non-negative integers a, b is $n \cdot m - n - m$.

A consequence of the theorem is that there are exactly $\frac{(n-1)(m-1)}{2}$ positive integers which cannot be expressed in the form $a \cdot n + b \cdot m$.

6.7 Bitwise operation

<ul style="list-style-type: none"> • $a + b = (a \oplus b) + 2(a \& b)$ • $a b = (a \oplus b) + (a \& b)$ • $a \& (b \oplus c) = (a \& b) \oplus (a \& c)$ • $a (b \& c) = (a b) \& (a c)$ • $a \& (b c) = (a \& b) (a \& c)$ • $a (a \& b) = a$ 	<ul style="list-style-type: none"> • $a \& (a b) = a$ • $n = 2^k \Leftrightarrow !(n \& (n - 1)) = 1$ • $-a = \sim a + 1$ • $4i \oplus (4i + 1) \oplus (4i + 2) \oplus (4i + 3) = 0$
--	--

Iterating over all subsets of a set and iterating over all sub-masks of a mask:

mask.h, 19 lines

```
int n;
void mask_example() {
    for (int mask = 0; mask < (1 << n); ++mask) {
        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) {
                // do something...
            }
        }
        // Time complexity: O(n * 2^n).
    }
    for (int mask = 0; mask < (1 << n); ++mask) {
        for (int submask = mask;;
            submask = (submask - 1) & mask) {
            // do something...
            if (submask == 0) { break; }
        }
        // Time complexity: O(3^n).
    }
}
```

6.8 Modmul

Description: calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.

Note: this runs roughly 2x faster than the naive `(...int128_t) a * b % M`.

Time: $O(1)$ for modmul, $O(\log b)$ for modpow.

modmul.h, 18 lines

```
#pragma once

uint64_t modmul(uint64_t a, uint64_t b, uint64_t mod) {
    int64_t ret =
        a * b -
        mod * uint64_t(1.L / mod * a * b); // overflow is
        fine!
    return ret + mod * (ret < 0) -
        mod * (ret >= (int64_t) mod);
}

uint64_t modpow(uint64_t a, uint64_t b, uint64_t mod) {
    uint64_t ans = 1;
    while (b > 0) {
        if (b & 1) { ans = modmul(ans, a, mod); }
        a = modmul(a, a, mod);
        b /= 2;
    }
}
```

```
    return ans;
}
```

6.9 Miller–Rabin

Description: Miller–Rabin primality test, this algorithm works for number up to $7e^{18}$.

miller_rabin.h, 28 lines

```
using num_t = long long;
int small_primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37, 73, 113, 193, 311, 313, 407521, 299210837};
bool miller_rabin(num_t a, num_t d, int s, num_t mod) {
    num_t x = modpow(a, d, mod);
    if (x == mod - 1 || x == 1) { return true; }
    for (int i = 0; i < s - 1; ++i) {
        x = modmul(x, x, mod);
        if (x == mod - 1) { return true; }
    }
    return false;
}
bool is_prime(num_t n) {
    if (n < 4) { return n > 1; }
    num_t d = n - 1;
    int s = 0;
    while (d % 2 == 0) {
        d >>= 1;
        s++;
    }
    for (int a : small_primes) {
        if (n == a) { return true; }
        if (n % a == 0 || !miller_rabin(a, d, s, n)) {
            return false;
        }
    }
    return true;
}
```

6.10 Pollard’s rho algorithm

Description: Pollard’s rho is an efficient algorithm for integer factorization. The algorithm can run smoothly with n upto $1e^{18}$, but be careful with overflow for larger n (e.g. $1e^{19}$).

pollard_rho.h, 54 lines

```
#include "miller_rabin.h"
#include "modmul.h"

uint64_t f(uint64_t x, int c,
    uint64_t mod) { // f(x) = (x^2 + c) % mod.
    x = modmul(x, x, mod) + c;
    if (x >= mod) { x -= mod; }
    return x;
}

uint64_t pollard_rho(uint64_t n, int c) {
    // algorithm to find a random divisor of 'n'.
    // using random function: f(x) = (x^2 + c) % n.
    uint64_t x = 2, y = x, d;
    long long p = 1;
    int dist = 0;
    while (true) {
        y = f(y, c, n);
        dist++;
        d = __gcd(max(x, y) - min(x, y), n);
        if (d > 1) { break; }
        if (dist == p) {
            dist = 0;
            p *= 2;
            x = y;
        }
    }
}
```

```
    return d;
}

void factorize(uint64_t n, vector<uint64_t>& factors) {
    if (n < 2) { return; }
    if (is_prime(n)) {
        factors.emplace_back(n);
        return;
    }
    uint64_t d = n;
    for (int c = 2; d == n; c++) { d = pollard_rho(n, c); }
    factorize(d, factors);
    factorize(n / d, factors);
}

vector<uint64_t> gen_divisors(
    vector<pair<uint64_t, int>>& factors) {
    vector<uint64_t> divisors = {1};
    for (auto& x : factors) {
        int sz = (int) divisors.size();
        for (int i = 0; i < sz; ++i) {
            uint64_t cur = divisors[i];
            for (int j = 0; j < x.second; ++j) {
                cur *= x.first;
                divisors.push_back(cur);
            }
        }
    }
    return divisors; // this array is NOT sorted yet.
}
```

6.11 Segment divisor sieve

Description: computes the number of divisors for each number in range $[L, R]$.

segment_divisor_sieve.h, 14 lines

```
const int MAXN = (int) 1e6; // R - L + 1 <= N.
int divisor_count[MAXN + 3];
void segment_divisor_sieve(long long L, long long R) {
    for (long long i = 1; i <= (long long) sqrt(R); ++i) {
        long long start1 = ((L + i - 1) / i) * i;
        long long start2 = i * i;
        long long j = max(start1, start2);
        if (j == start2) {
            divisor_count[j - L] += 1;
            j += i;
        }
        for (; j <= R; j += i) { divisor_count[j - L] += 2; }
    }
}
```

6.12 Linear sieve

Description: finding primes and computing value for multiplicative function in $O(N)$.

Time: $O(N)$ (but the factor may be large).

linear_sieve.h, 46 lines

```
const int N = (int) 2e6 + 3;
bool is_prime[N + 1];
int spf[N + 1]; // smallest prime factor
int lpf[N + 1]; // largest prime factor
int cntp[N + 1]; // number of prime factor
int phi[N + 1]; // euler’s totient function
int mu[N + 1]; // mobius function
int func[N + 1]; // a multiplicative function, f(p^k) = k
int k[N + 1]; // k[i] = the power of the smallest prime
                // factor of i
int pw[N + 1]; // pw[i] = p^k[i] where p is the smallest
                // prime factor of i
vector<int> primes;
```

```
void linear_sieve(int n = N) {
    spf[0] = spf[1] = lpf[0] = lpf[1] = -1;
    phi[1] = mu[1] = func[1] = 1;
    for (int x = 2; x <= n; ++x) {
        if (spf[x] == 0) {
            primes.push_back(x);
            is_prime[x] = true;
            spf[x] = lpf[x] = x;
            cntp[x] = 1;
            phi[x] = x - 1, mu[x] = -1, func[x] = 1;
            k[x] = 1, pw[x] = x;
        }
        for (int p : primes) {
            if (p > spf[x] || x * p > n) { break; }
            spf[x * p] = p, lpf[x * p] = lpf[x];
            cntp[x * p] = cntp[x] + 1;
            if (p == spf[x]) {
                phi[x * p] = phi[x] * p;
                mu[x * p] = 0;
                func[x * p] = func[x / pw[x]] * (k[x] + 1);
                k[x * p] = k[x] + 1;
                pw[x * p] = pw[x] * p;
            } else {
                phi[x * p] = phi[x] * phi[p];
                mu[x * p] = mu[x] * mu[p]; // or -mu[x]
                func[x * p] = func[x] * func[p];
                k[x * p] = 1;
                pw[x * p] = p;
            }
        }
    }
}
```

6.13 Bitset sieve

Description: sieve of eratosthenes for large n (up to 10^9).

Time: time and space tested on codeforces:

- For $n = 10^8$: 200 ms, 6 MB.
- For $n = 10^9$: 4000 ms, 60 MB.

bitset_sieve.h, 21 lines

```
const int N = (int) 1e8;
bitset<N / 2 + 1> isPrime;
void sieve(int n = N) {
    isPrime.flip();
    isPrime[0] = false;
    for (int i = 3; i <= (int) sqrt(n); i += 2) {
        if (isPrime[i >> 1]) {
            for (int j = i * i; j <= n; j += 2 * i) {
                isPrime[j >> 1] = false;
            }
        }
    }
}

void example(int n) {
    sieve(n);
    int primeCnt = (n >= 2);
    for (int i = 3; i <= n; i += 2) {
        if (isPrime[i >> 1]) { primeCnt++; }
    }
    cout << primeCnt << '\n';
}
```

6.14 Block sieve

Description: a very fast sieve of eratosthenes for large n (up to 10^9).

Time: time and space tested on codeforces:

- For $n = 10^8$: 160 ms, 60 MB.

- For $n = 10^9$: 1600 ms, 505 MB.

block_sieve.h, 29 lines

```
const int N = (int) 1e8;
bitset<N + 1> is_prime;
vector<int> fast_sieve() {
    const int S = (int) sqrt(N), R = N / 2;
    vector<int> primes = {2};
    vector<bool> sieve(S + 1, true);
    vector<array<int, 2>> cp;
    for (int i = 3; i <= S; i += 2) {
        if (sieve[i]) {
            cp.push_back({i, i * i / 2});
            for (int j = i * i; j <= S; j += 2 * i) {
                sieve[j] = false;
            }
        }
    }
    for (int L = 1; L <= R; L += S) {
        array<bool, S> block{};
        for (auto& [p, idx] : cp) {
            for (; idx < S + L; idx += p) {
                block[idx - L] = true;
            }
        }
        for (int i = 0; i < min(S, R - L); ++i) {
            if (!block[i]) { primes.push_back((L + i) * 2 + 1); }
        }
    }
    for (int p : primes) { is_prime[p] = true; }
    return primes;
}
```

6.15 Sqrt mod

Description: Tonelli-Shanks algorithm. For a given non-negative integer a and a prime number p , find x such that $x^2 \equiv a \pmod{p}$ or -1 if there is no such x .

sqrt_mod.h, 30 lines

```
#include "<math>./modmul.h</math>"

int mod_sqrt(int a, int p) {
    if (a == 0) { return 0; }
    if (p == 2) { return (a & 1 ? 1 : 0); }
    if (modpow(a, (p - 1) / 2, p) != 1) { return -1; }

    int b = 1;
    while (modpow(b, (p - 1) / 2, p) == 1) { ++b; }
    int d = p - 1, e = 0; // p - 1 = d * 2^s
    while (d % 2 == 0) { d /= 2, ++e; }
    int64_t x = modpow(a, (d - 1) / 2, p);
    int64_t y = a * x % p * x % p;
    x = x * a % p;
    int64_t z = modpow(b, d, p);
    while (y != 1) {
        int i = 0;
        int64_t k = y;
        while (k != 1) {
            ++i;
            k = k * k % p;
        }
        z = modpow(z, 1 << (e - i - 1), p);
        x = x * z % p;
        z = z * z % p;
        y = y * z % p;
        e = i;
    }
    return x;
}
```

}

6.16 Extended Euclidean

Description: for two integers a and b , extended Euclidean algorithm allows computing x and y such that: $ax + by = \gcd(a, b)$. Note that such representation always exists by Bézout’s identity.

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

extended_euclidean.h, 13 lines

```
template<typename T>
T extended_euclidean(T a, T b, T& x, T& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    T x1, y1;
    T d = extended_euclidean(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

7 Combinatorics

7.1 Catalan numbers

$$C_n = \frac{1}{n + 1} \binom{2n}{n} = \frac{(2n)!}{n!(n + 1)!}$$

$$C_{n+1} = \sum_{i=0}^n C_i C_{n-i}, \quad C_0 = 1, \quad C_n = \frac{4n - 2}{n + 1} C_{n-1}$$

n	0	1	2	3	4	5	6	7	8
C_n	1	1	2	5	14	42	132	429	1430
n	9	10	11	12	13				
C_n	4862	16796	58786	208012	742900				

Applications of Catalan numbers:

- difference binary search trees with n vertices from 1 to n .
- rooted binary trees with $n + 1$ leaves (vertices are not numbered).
- correct bracket sequence of length $2 * n$.
- permutation $[n]$ with no 3-term increasing subsequence (i.e. doesn’t exist $i < j < k$ for which $a[i] < a[j] < a[k]$).
- ways a convex polygon of $n + 2$ sides can split into triangles by connecting vertices.

7.2 Fibonacci numbers

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{otherwise} \end{cases}$$

- The first 20 Fibonacci numbers ($n = 0, 1, 2, \dots, 19$):

$$F_n = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, \dots$$

- Binet’s formula:

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}}$$

$$\text{where } \varphi = \frac{1 + \sqrt{5}}{2}, \psi = \frac{1 - \sqrt{5}}{2}$$

- Properties:

$$\begin{aligned} F_{2n+1} &= F_n^2 + F_{n+1}^2 & \left| \begin{aligned} F_{n+1} \cdot F_{n-1} - F_n^2 &= (-1)^n \\ n \mid m &\Leftrightarrow F_n \mid F_m \\ \gcd(F_n, F_m) &= F_{\gcd(n, m)} \end{aligned} \right. \\ F_{2n} &= F_{n-1} \cdot F_n + F_n \cdot F_{n+1} \end{aligned}$$

7.3 Stirling numbers of the first kind

Number of permutations of n elements which contain exactly k permutation cycles.

$$S(0, 0) = 1$$

$$S(n, k) = S(n - 1, k - 1) + (n - 1)S(n - 1, k)$$

$$\sum_{k=0}^n S(n, k) x^k = x(x + 1)(x + 2) \dots (x + n - 1)$$

7.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k non-empty groups.

$$S(n, 1) = S(n, n) = 1$$

$$S(n, k) = S(n - 1, k - 1) + kS(n - 1, k)$$

$$S(n, k) = \frac{1}{k!} \sum_{i=0}^k (-1)^{k-i} \binom{k}{i} i^n$$

7.5 Derangements

Permutation of the elements of a set, such that no element appears in its original position (no fixed point). Recursive formulas:

$$D(n) = (n - 1)[D(n - 1) + D(n - 2)] = nD(n - 1) + (-1)^n$$

8 Geometry

8.1 Fundamentals

8.1.1 Point

point.h, 86 lines

```
#pragma once

const double PI = acos(-1);
const double EPS = 1e-9;
typedef double ftype;
struct Point {
    ftype x, y;
    Point(ftype _x = 0, ftype _y = 0): x(_x), y(_y) {}
    Point& operator+=(const Point& other) {
        x += other.x;
        y += other.y;
        return *this;
    }
    Point& operator-=(const Point& other) {
        x -= other.x;
        y -= other.y;
        return *this;
    }
    Point& operator*=(ftype t) {
        x *= t;
        y *= t;
        return *this;
    }
    Point& operator/=(ftype t) {
        x /= t;
        y /= t;
        return *this;
    }
    Point operator+(const Point& other) const {
        return Point(*this) += other;
    }
    Point operator-(const Point& other) const {
        return Point(*this) -= other;
    }
    Point operator*(ftype t) const {
        return Point(*this) *= t;
    }
    Point operator/(ftype t) const {
        return Point(*this) /= t;
    }
    Point rotate(double angle) const {
        return Point(x * cos(angle) - y * sin(angle),
            x * sin(angle) + y * cos(angle));
    }
    friend istream& operator>>(istream& in, Point& t);
    friend ostream& operator<<(ostream& out, const Point& t);
    bool operator<(const Point& other) const {
        if (fabs(x - other.x) < EPS) { return y < other.y; }
        return x < other.x;
    }
};

istream& operator>>(istream& in, Point& t) {
    in >> t.x >> t.y;
    return in;
}

ostream& operator<<(ostream& out, const Point& t) {
    out << t.x << ' ' << t.y;
    return out;
}
```

```
ftype dot(Point a, Point b) {
    return a.x * b.x + a.y * b.y;
}
ftype norm(Point a) { return dot(a, a); }
ftype abs(Point a) { return sqrt(norm(a)); }
ftype angle(Point a, Point b) {
    return acos(dot(a, b) / (abs(a) * abs(b)));
}
ftype proj(Point a, Point b) { return dot(a, b) / abs(b); }
ftype cross(Point a, Point b) {
    return a.x * b.y - a.y * b.x;
}
bool ccw(Point a, Point b, Point c) {
    return cross(b - a, c - a) > EPS;
}
int sign(ftype val) {
    return (val < -EPS ? -1 : val >= EPS ? 1 : 0);
}
bool collinear(Point a, Point b, Point c) {
    return fabs(cross(b - a, c - a)) < EPS;
}
Point intersect(Point a1, Point d1, Point a2, Point d2) {
    double t = cross(a2 - a1, d2) / cross(d1, d2);
    return a1 + d1 * t;
}
```

8.1.2 Line

line.h, 93 lines

```
#include "point.h"

struct Line {
    double a, b, c;
    Line(double _a = 0, double _b = 0, double _c = 0):
        a(_a), b(_b), c(_c) {}
    friend ostream& operator<<(ostream& out, const Line& l);
};

ostream& operator<<(ostream& out, const Line& l) {
    out << l.a << ' ' << l.b << ' ' << l.c;
    return out;
}

void PointsToLine(
    const Point& p1, const Point& p2, Line& l) {
    if (fabs(p1.x - p2.x) < EPS) {
        l = {1.0, 0.0, -p1.x};
    } else {
        l.a = -(double) (p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0;
        l.c = -l.a * p1.x - l.b * p1.y;
    }
}

void PointsSlopeToLine(const Point& p, double m, Line& l) {
    l.a = -m;
    l.b = 1;
    l.c = -l.a * p.x - l.b * p.y;
}

bool areParallel(const Line& l1, const Line& l2) {
    return fabs(l1.a - l2.a) < EPS && fabs(l1.b - l2.b) < EPS;
}

bool areSame(const Line& l1, const Line& l2) {
    return areParallel(l1, l2) && fabs(l1.c - l2.c) < EPS;
}

bool areIntersect(Line l1, Line l2, Point& p) {
    if (areParallel(l1, l2)) { return false; }
    p.x = -(l1.c * l2.b - l1.b * l2.c) /
        (l1.a * l2.b - l1.b * l2.a);
    if (fabs(l1.b) > EPS) {
```

```
        p.y = -(l1.c + l1.a * p.x);
    } else {
        p.y = -(l2.c + l2.a * p.x);
    }
    return l1;
}

double distToLine(Point p, Point a, Point b, Point& c) {
    double t = dot(p - a, b - a) / norm(b - a);
    c = a + (b - a) * t;
    return abs(c - p);
}

double distToSegment(Point p, Point a, Point b, Point& c) {
    double t = dot(p - a, b - a) / norm(b - a);
    if (t > 1.0) {
        c = Point(b.x, b.y);
    } else if (t < 0.0) {
        c = Point(a.x, a.y);
    } else {
        c = a + (b - a) * t;
    }
    return abs(c - p);
}

bool intersectTwoSegment(
    Point a, Point b, Point c, Point d) {
    ftype ABxAC = cross(b - a, c - a);
    ftype ABxAD = cross(b - a, d - a);
    ftype CDxCA = cross(d - c, a - c);
    ftype CDxCB = cross(d - c, b - c);
    if (ABxAC == 0 || ABxAD == 0 || CDxCA == 0 || CDxCB == 0) {
        if (ABxAC == 0 && dot(a - c, b - c) <= 0) {
            return true;
        }
        if (ABxAD == 0 && dot(a - d, b - d) <= 0) {
            return true;
        }
        if (CDxCA == 0 && dot(c - a, d - a) <= 0) {
            return true;
        }
        if (CDxCB == 0 && dot(c - b, d - b) <= 0) {
            return true;
        }
        return false;
    }
    return (ABxAC * ABxAD < 0 && CDxCA * CDxCB < 0);
}

void perpendicular(Line l1, Point p, Line& l2) {
    if (fabs(l1.a) < EPS) {
        l2 = {1.0, 0.0, -p.x};
    } else {
        l2.a = -l1.b / l1.a;
        l2.b = 1.0;
        l2.c = -l2.a * p.x - l2.b * p.y;
    }
}
```

8.1.3 Circle

circle.h, 19 lines

```
#include "point.h"

int insideCircle(
    const Point& p, const Point& center, ftype r) {
    ftype d = norm(p - center);
    ftype rSq = r * r;
    return fabs(d - rSq) < EPS ? 0
        : (d - rSq >= EPS ? 1 : -1);
}

bool circle2PointsR(
```

```

const Point& p1, const Point& p2, ftype r, Point& c) {
double h = r * r - norm(p1 - p2) / 4.0;
if (fabs(h) < 0) { return false; }
h = sqrt(h);
Point perp = (p2 - p1).rotate(PI / 2.0);
Point m = (p1 + p2) / 2.0;
c = m + perp * (h / abs(perp));
return true;
}

```

8.1.4 Triangle

triangle.h, 35 lines

```

#include "line.h"
#include "point.h"

double areaTriangle(double ab, double bc, double ca) {
double p = (ab + bc + ca) / 2;
return sqrt(p) * sqrt(p - ab) * sqrt(p - bc) *
sqrt(p - ca);
}

double rInCircle(double ab, double bc, double ca) {
double p = (ab + bc + ca) / 2;
return areaTriangle(ab, bc, ca) / p;
}

double rInCircle(Point a, Point b, Point c) {
return rInCircle(abs(a - b), abs(b - c), abs(c - a));
}

bool inCircle(
Point p1, Point p2, Point p3, Point& ctr, double& r) {
r = rInCircle(p1, p2, p3);
if (fabs(r) < EPS) { return false; }
line l1, l2;
double ratio = abs(p2 - p1) / abs(p3 - p1);
Point p = p2 + (p3 - p2) * (ratio / (1 + ratio));
PointsToLine(p1, p, l1);
ratio = abs(p1 - p2) / abs(p2 - p3);
p = p1 + (p3 - p1) * (ratio / (1 + ratio));
PointsToLine(p2, p, l2);
areIntersect(l1, l2, ctr);
return true;
}

double rCircumCircle(double ab, double bc, double ca) {
return ab * bc * ca / (4.0 * areaTriangle(ab, bc, ca));
}

double rCircumCircle(Point a, Point b, Point c) {
return rCircumCircle(abs(b - a), abs(c - b), abs(a -
c));
}

```

8.1.5 Convex hull

Description: Andrew's algorithm for computing convex hull of a set of points.

Time: $O(n \log n)$

convex_hull.h, 24 lines

```

#include "point.h"

vector<Point> convex_hull(vector<Point>&& points) {
int n = (int) points.size(), k = 0;
if (n <= 2) { return points; }
vector<Point> ch(n * 2);
sort(points.begin(), points.end());
for (int i = 0; i < n; ++i) {
while (k >= 2 && sign(cross(ch[k - 1] - ch[k - 2],
points[i] - ch[k - 1])) <= -1) {
--k;
}
ch[k++] = points[i];
}
}

```

```

for (int i = n - 2, t = k + 1; i >= 0; --i) {
while (k >= t && sign(cross(ch[k - 1] - ch[k - 2],
points[i] - ch[k - 1])) <= -1) {
--k;
}
ch[k++] = points[i];
}
ch.resize(k - 1);
return ch;
}

```

8.1.6 Polygon

polygon.h, 49 lines

```

#include "point.h"

double perimeter(const vector<Point>& P) {
double ans = 0.0;
for (int i = 0; i < (int) P.size() - 1; ++i) {
ans += abs(P[i] - P[i + 1]);
}
return ans;
}

double area(const vector<Point>& P) {
double ans = 0.0;
for (int i = 0; i < (int) P.size() - 1; ++i) {
ans += (P[i].x * P[i + 1].y - P[i + 1].x * P[i].y);
}
return fabs(ans) / 2.0;
}

bool isConvex(const vector<Point>& P) {
int n = (int) P.size();
if (n <= 3) { return false; }
bool firstTurn = ccw(P[0], P[1], P[2]);
for (int i = 1; i < n - 1; ++i) {
if (ccw(P[i], P[i + 1], P[i + 2]) == n ? 1 : i + 2))
!= firstTurn) {
return false;
}
}
return true;
}

int insidePolygon(Point pt, const vector<Point>& P) {
int n = (int) P.size();
if (n <= 3) { return -1; }
bool on_polygon = false;
for (int i = 0; i < n - 1; ++i) {
if (fabs(abs(P[i] - pt) + abs(pt - P[i + 1]) -
abs(P[i] - P[i + 1])) < EPS) {
on_polygon = true;
}
}
if (on_polygon) { return 0; }
double sum = 0.0;
for (int i = 0; i < n - 1; ++i) {
if (ccw(pt, P[i], P[i + 1])) {
sum += angle(P[i] - pt, P[i + 1] - pt);
} else {
sum -= angle(P[i] - pt, P[i + 1] - pt);
}
}
return fabs(sum) > PI ? 1 : -1;
}

```

8.2 KD tree

Description: KD-tree searching for closest point to the given point, can also be changed to find farthest point.

Time: average-case complexity is $O(3^d \log N)$ where d is the number of dimensions

kd_tree.h, 110 lines

```

using T = long long;
const T INF = numeric_limits<T>::max();
struct Point {
T x, y;
Point(T _x = 0, T _y = 0): x(_x), y(_y) {}
T dist(const Point& other) const {
T dx = x - other.x, dy = y - other.y;
return dx * dx + dy * dy;
}
bool operator<(const Point& other) const {
return tie(x, y) < tie(other.x, other.y);
}
bool operator==(const Point& other) const {
return tie(x, y) == tie(other.x, other.y);
}
};

bool comp_x(const Point& a, const Point& b) {
return a.x < b.x;
}

bool comp_y(const Point& a, const Point& b) {
return a.y < b.y;
}

struct Node {
Point point; // a single point if this Node is a leaf
T x_left = INF, x_right = -INF, y_left = INF,
y_right = -INF;
Node *first = nullptr,
*second = nullptr; // two children of this node
T dist(const Point& A) {
// MIN squared distance between the point A and this
// box, 0 if inside to compute MAX distance, calculate
// MAX distance from A to the four corner points of
this
// box
T x = (A.x < x_left ? x_left
: A.x > x_right ? x_right
: A.x);
T y = (A.y < y_left ? y_left
: A.y > y_right ? y_right
: A.y);
return A.dist(Point(x, y));

// MAX squared distance
// T x, y;
// if (A.x < x_left) x = x_right;
// else if (A.x > x_right) x = x_left;
// else x = A.x - x_left > x_right - A.x ? x_left :
// x_right;

// if (A.y < y_left) y = y_right;
// else if (A.y > y_right) y = y_left;
// else y = A.y - y_left > y_right - A.y ? y_left :
// y_right;
return A.dist(Point(x, y));
}
Node(vector<Point>&& points): point(points[0]) {
for (auto& p : points) {
x_left = min(x_left, p.x);
x_right = max(x_right, p.x);
y_left = min(y_left, p.y);
y_right = max(y_right, p.y);
}
int sz = (int) points.size();
if (sz > 1) {

```

```

    // split on x if width >= height (not ideal...)
    sort(points.begin(), points.end(),
        x_right - x_left >= y_right - y_left ? comp_x
        : comp_y);
    // divide by taking half the array for each child
    (not
    // best performance with many duplicates in the
    // middle)
    int half = sz / 2;
    first =
    new Node({points.begin(), points.begin() + half});
    second =
    new Node({points.begin() + half, points.end()});
}
}
};

struct KDTree {
    Node *root;
    KDTree(const vector<Point>& points):
        root(new Node({points.begin(), points.end()})) {}
    pair<T, Point> search(Node *node, const Point& point) {
        if (!node->first) {
            // uncomment if we SHOULD NOT find the point itself
            // if (node->point == point) return pair{INF,
            // Point{}};
            return pair{point.dist(node->point), node->point};
        }
        Node *first = node->first, *second = node->second;
        T bfirst = first->dist(point),
        bsecond = second->dist(point);
        if (bfirst > bsecond) {
            swap(bfirst, bsecond), swap(first, second);
        }

        // search closest side first, other side if needed
        auto best = search(first, point);
        if (bsecond < best.first) {
            best = min(best, search(second, point));
        }
        return best;
    }
    pair<T, Point> search(const Point& point) {
        return search(root, point);
    }
};

```

9 Linear algebra

9.1 Gauss elimination

Time: $O(\min(n, m) \cdot nm)$ or $O(n^3)$ in case $n = m$.

gauss_elimination.h, 44 lines

```

const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be
// infinity or a big number
int gauss(vector<vector<double>> a, vector<double>& ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;
    vector<int> where(m, -1);
    for (int col = 0, row = 0; col < m && row < n; ++col) {
        int sel = row;
        for (int i = row; i < n; ++i) {
            if (abs(a[i][col]) > abs(a[sel][col])) { sel = i; }
        }
        if (abs(a[sel][col]) < EPS) { continue; }
        for (int i = col; i <= m; ++i) {

```

```

            swap(a[sel][i], a[row][i]);
        }
        where[col] = row;
        for (int i = 0; i < n; ++i) {
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j = col; j <= m; ++j) {
                    a[i][j] -= a[row][j] * c;
                }
            }
        }
        ++row;
    }
    ans.assign(m, 0);
    for (int i = 0; i < m; ++i) {
        if (where[i] != -1) {
            ans[i] = a[where[i]][m] / a[where[i]][i];
        }
    }
    for (int i = 0; i < n; ++i) {
        double sum = 0;
        for (int j = 0; j < m; ++j) { sum += ans[j] *
            a[i][j]; }
        if (abs(sum - a[i][m]) > EPS) { return 0; }
    }
    for (int i = 0; i < m; ++i) {
        if (where[i] == -1) { return INF; }
    }
    return 1;
}

```

9.2 Gauss determinant

Description: computing determinant of a square matrix A by applying Gauss elimination to produces a row echolon matrix B, then the determinant of A is equal to product of the elements of the diagonal of B.

Time: $O(N^3)$.

gauss_determinant.h, 30 lines

```

const double EPS = 1e-9;
double determinant(vector<vector<double>> A) {
    int n = (int) A.size();
    double det = 1;
    for (int i = 0; i < n; ++i) {
        // find non-zero cell
        int k = i;
        for (int j = i + 1; j < n; ++j) {
            if (abs(A[j][i]) > abs(A[k][i])) { k = j; }
        }
        if (abs(A[k][i]) < EPS) {
            det = 0;
            break;
        }
        if (i != k) {
            swap(A[i], A[k]);
            det = -det;
        }
        det *= A[i][i];
        for (int j = i + 1; j < n; ++j) { A[i][j] /= A[i][i]; }
        for (int j = 0; j < n; ++j) {
            if (j != i && abs(A[j][i]) > EPS) {
                for (int k = i + 1; k < n; ++k) {
                    A[j][k] -= A[i][k] * A[j][i];
                }
            }
        }
    }
}

```

```

    return det;
}

```

9.3 Bareiss determinant

Description: Bareiss algorithm for computing determinant of a square matrix A with integer entries using only integer arithmetic.

Usage:

- Kirchhoff's theorem: finding the number of spanning trees.

Time: $O(N^3)$.

bareiss_determinant.h, 31 lines

```

long long determinant(vector<vector<long long>> A) {
    int n = (int) A.size();
    long long prev = 1;
    int sign = 1;
    for (int i = 0; i < n - 1; ++i) {
        // find non-zero cell
        if (A[i][i] == 0) {
            int k = -1;
            for (int j = i + 1; j < n; ++j) {
                if (A[j][i] != 0) {
                    k = j;
                    break;
                }
            }
            if (k == -1) { return 0; }
            swap(A[i], A[k]);
            sign = -sign;
        }
        for (int j = i + 1; j < n; ++j) {
            for (int k = i + 1; k < n; ++k) {
                assert(
                    (A[j][k] * A[i][i] - A[j][i] * A[i][k]) % prev
                    ==
                    0);
                A[j][k] =
                    (A[j][k] * A[i][i] - A[j][i] * A[i][k]) / prev;
            }
        }
        prev = A[i][i];
    }
    return sign * A[n - 1][n - 1];
}

```

10 Graph

10.1 Bellman-Ford algorithm

Description: single source shortest path in a weighted (negative or positive) directed graph.

Time: $O(VE)$.

bellman_ford.h, 36 lines

```

const int64_t INF = (int64_t) 2e18;
struct Edge {
    int u, v; // u -> v
    int64_t w;
    Edge() {}
    Edge(int _u, int _v, int64_t _w): u(_u), v(_v), w(_w) {}
};
int n;
vector<Edge> edges;
vector<int64_t> bellmanFord(int s) {
    // dist[stating] = 0.
    // dist[u] = +INF, if u is unreachable.
    // dist[u] = -INF, if there is a negative cycle on the
    // path from s to u. -INF < dist[u] < +INF, otherwise.

```

```
vector<int64_t> dist(n, INF);
dist[s] = 0;
for (int i = 0; i < n - 1; ++i) {
    bool any = false;
    for (auto [u, v, w] : edges) {
        if (dist[u] != INF && dist[v] > w + dist[u]) {
            dist[v] = w + dist[u];
            any = true;
        }
    }
    if (!any) { break; }
}
// handle negative cycles
for (int i = 0; i < n - 1; ++i) {
    for (auto [u, v, w] : edges) {
        if (dist[u] != INF && dist[v] > w + dist[u]) {
            dist[v] = -INF;
        }
    }
}
return dist;
}
```

10.2 Articulation point and Bridge

Description: finding articulation points and bridges in a simple undirected graph.

Time: $O(V + E)$.

articulation_point_and_bridge.h, 41 lines

```
const int N = (int) 1e5;
vector<int> g[N];
int num[N], low[N], dfs_timer;
bool joint[N];
vector<pair<int, int>> bridges;
void dfs(int u, int prev = -1) {
    low[u] = num[u] = ++dfs_timer;
    int child = 0;
    for (int v : g[u]) {
        if (v == prev) { continue; }
        if (num[v]) {
            low[u] = min(low[u], num[v]);
        } else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            child++;
            if (low[v] >= num[u]) { bridges.emplace_back(u, v); }
            if (prev != -1 && low[v] >= num[u]) {
                joint[u] = true;
            }
        }
    }
    if (prev == -1 && child > 1) { joint[u] = true; }
}

int solve() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        u--;
        v--;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for (int i = 0; i < n; ++i) {
        if (!num[i]) { dfs(i); }
    }
}
```

```
    return 0;
}
```

10.3 Topo sort

Description: a topological sort of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge from vertex u to vertex v , u comes before v in the ordering.

Note: if there are cycles, the returned list will have size smaller than n .

Time: $O(V + E)$.

topo_sort.h, 22 lines

```
vector<int> topo_sort(const vector<vector<int>>& g) {
    int n = (int) g.size();
    vector<int> indeg(n);
    for (int u = 0; u < n; ++u) {
        for (int v : g[u]) { indeg[v]++; }
    }
    queue<int> q; // Note: use min-heap to get the smallest
                  // lexicographical order.
    for (int u = 0; u < n; ++u) {
        if (indeg[u] == 0) { q.emplace(u); }
    }
    vector<int> topo;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        topo.emplace_back(u);
        for (int v : g[u]) {
            if (--indeg[v] == 0) { q.emplace(v); }
        }
    }
    return topo;
}
```

10.4 Strongly connected components

10.4.1 Tarjan's Algorithm

Description: Tarjan's algorithm finds strongly connected components (SCC) in a directed graph. If two vertices u and v belong to the same component, then $scc_id[u] == scc_id[v]$.

Time: $O(V + E)$.

tarjan.h, 27 lines

```
const int N = (int) 5e5;
vector<int> g[N], st;
int low[N], num[N], dfs_timer, scc_id[N], scc;
bool used[N];
void Tarjan(int u) {
    low[u] = num[u] = ++dfs_timer;
    st.push_back(u);
    for (int v : g[u]) {
        if (used[v]) { continue; }
        if (num[v] == 0) {
            Tarjan(v);
            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], num[v]);
        }
    }
    if (low[u] == num[u]) {
        int v;
        do {
            v = st.back();
            st.pop_back();
            used[v] = true;
            scc_id[v] = scc;
        } while (v != u);
    }
}
```

```
    scc++;
}
}
```

10.4.2 Kosaraju's algorithm

Description: Kosaraju's algorithm finds strongly connected components (SCC) in a directed graph in a straightforward way. Two vertices u and v belong to the same component iff $scc_id[u] == scc_id[v]$. This algorithm generates connected components numbered in topological order in corresponding condensation graph.

Time: $O(V + E)$.

kosaraju.h, 36 lines

```
const int N = (int) 1e5;
vector<int> g[N], rev_g[N], vers;
int scc_id[N];
bool vis[N];
int n, m;

void dfs1(int u) {
    vis[u] = true;
    for (int v : g[u]) {
        if (!vis[v]) { dfs1(v); }
    }
    vers.push_back(u);
}

void dfs2(int u, int id) {
    scc_id[u] = id;
    vis[u] = true;
    for (int v : rev_g[u]) {
        if (!vis[v]) { dfs2(v, id); }
    }
}

void Kosaraju() {
    for (int i = 0; i < n; ++i) {
        if (!vis[i]) { dfs1(i); }
    }
    memset(vis, 0, sizeof vis);
    int scc_cnt = 0;
    // iterating in reverse order
    for (int i = n - 1; i >= 0; --i) {
        int u = vers[i];
        if (!vis[u]) { dfs2(u, ++scc_cnt); }
    }
    cout << scc_cnt << '\n';
    for (int i = 0; i < n; ++i) {
        cout << scc_id[i] << " \n"[i == n - 1];
    }
}
```

10.5 K-th smallest shortest path

Description: finding the k -th smallest shortest path from vertex s to vertex t , each vertex can be visited more than once.

k_smallest_shortest_path.h, 21 lines

```
using adj_list = vector<vector<pair<int, int>>>;
vector<long long> k_smallest(
    const adj_list& g, int k, int s, int t) {
    int n = (int) g.size();
    vector<long long> ans;
    vector<int> cnt(n);
    using pli = pair<long long, int>;
    priority_queue<pli, vector<pli>, greater<pli>> pq;
    pq.emplace(0, s);
    while (!pq.empty() && cnt[t] < k) {
        int u = pq.top().second;
        long long d = pq.top().first;
```

```
    pq.pop();
    if (cnt[u] == k) { continue; }
    cnt[u]++;
    if (u == t) { ans.push_back(d); }
    for (auto [v, cost] : g[u]) { pq.emplace(d + cost, v); }
}
assert(k == (int) ans.size());
return ans;
}
```

10.6 Eulerian path

10.6.1 Directed graph

Description: Hierholzer’s algorithm. An Eulerian path in a directed graph is a path that visits all edges exactly once. An Eulerian cycle is a Eulerian path that is a cycle.
Time: $O(E)$.

eulerian_path_directed.h, 17 lines

```
vector<int> find_path_directed(
    const vector<vector<int>>& g, int s) {
    int n = (int) g.size();
    vector<int> stack, cur_edge(n), vertices;
    stack.push_back(s);
    while (!stack.empty()) {
        int u = stack.back();
        stack.pop_back();
        while (cur_edge[u] < (int) g[u].size()) {
            stack.push_back(u);
            u = g[u][cur_edge[u]++];
        }
        vertices.push_back(u);
    }
    reverse(vertices.begin(), vertices.end());
    return vertices;
}
```

10.6.2 Undirected graph

Description: Hierholzer’s algorithm. An Eulerian path in a undirected graph is a path that visits all edges exactly once. An Eulerian cycle is a Eulerian path that is a cycle.
Time: $O(E)$.

eulerian_path_undirected.h, 21 lines

```
struct Edge {
    int to;
    list<Edge>::iterator reverse_edge;
    Edge(int _to): to(_to) {}
};
vector<int> vertices;
void find_path(vector<list<Edge>>& g, int u) {
    while (!g[u].empty()) {
        int v = g[u].front().to;
        g[v].erase(g[u].front().reverse_edge);
        g[u].pop_front();
        find_path(g, v);
    }
    vertices.emplace_back(u); // reversion list.
}
void add_edge(vector<list<Edge>>& g, int u, int v) {
    g[u].emplace_front(v);
    g[v].emplace_front(u);
    g[u].front().reverse_edge = g[v].begin();
    g[v].front().reverse_edge = g[u].begin();
}
```

10.7 Network flow

10.7.1 Flow

flow.h, 55 lines

```
const int N = (int) 1e3 + 3;
const int oo = (int) 1e9;
int trace[N], c[N][N], f[N][N];
vector<int> adj[N];
int n, s, t;

bool FindPath() {
    for (int u = 1; u <= n; ++u) { trace[u] = 0; }

    queue<int> q;
    q.push(s);
    trace[s] = s;

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

        for (int v : adj[u]) {
            if (!trace[v] && c[u][v] > f[u][v]) {
                trace[v] = u;
                if (v == t) { return 1; }
                q.push(v);
            }
        }
    }
    return 0;

void Enlarge() {
    int u, v = t, mn = oo;
    while (v != s) {
        u = trace[v];
        mn = min(mn, c[u][v] - f[u][v]);
        v = u;
    }

    v = t;
    while (v != s) {
        u = trace[v];
        f[u][v] += mn;
        f[v][u] -= mn;
        v = u;
    }
}

int solve() {
    // Xu ly dau vao

    while (FindPath()) { Enlarge(); }

    int ans = 0;
    for (int u = 1; u <= n; ++u) { ans += f[u][t]; }
    cout << ans << '\n';
    return 0;
}
```

10.7.2 Min cost max flow

min_cost_max_flow.h, 65 lines

```
const int N = (int) 1e3 + 3;
const int oo = (int) 1e9;
int trace[N], c[N][N], f[N][N], d[N];
vector<pair<int, int>> adj[N];
int n, s, t, ans;

bool FindPath() {
    for (int u = 1; u <= n; ++u) {
```

```
        trace[u] = -1;
        d[u] = oo;
    }

    trace[s] = s;
    d[s] = 0;

    using Node = pair<int, int>;
    priority_queue<Node, vector<Node>, greater<Node>> pq;
    pq.push({0, s});
    while (!pq.empty()) {
        auto [l, u] = pq.top();
        pq.pop();

        if (l > d[u]) { continue; }
        for (auto [w, v] : adj[u]) {
            if (c[u][v] > f[u][v]) {
                if (l + (f[u][v] >= 0 ? w : -w) < d[v]) {
                    d[v] = l + (f[u][v] >= 0 ? w : -w);
                    trace[v] = u;

                    pq.push({d[v], v});
                }
            }
        }
    }

    return trace[t] != -1;
}

void Enlarge() {
    int u, v = t, mn = oo;
    while (v != s) {
        u = trace[v];
        mn = min(mn, c[u][v] - f[u][v]);
        v = u;
    }

    v = t;
    while (v != s) {
        u = trace[v];
        f[u][v] += mn;
        f[v][u] -= mn;
        v = u;
    }

    ans += d[t] * mn;
}

int solve() {
    // Xu ly dau vao

    while (FindPath()) { Enlarge(); }

    cout << ans << '\n';
    return 0;
}
```

10.8 Trees

10.8.1 LCA

Description: finding lowest common ancestor (LCA) between any two vertices.
Time: $< O(N \log N), O(1) >$.

lca.h, 34 lines

```
#include "../data-structures/rmq.h"

struct LCA {
    int n;
    vector<int> pos, depth;
    vector<vector<int>> g;
```



```
vector<pair<int, int>> tour;
RMQ<pair<int, int>> rmq;
LCA(int _n): n(_n), pos(n), depth(n), g(n) {}
void add_edge(int u, int v) { g[u].emplace_back(v); }
void build(int root = 0) {
    dfs(root);
    rmq.build(tour);
}
void dfs(int u, int par = -1) {
    pos[u] = (int) tour.size();
    tour.emplace_back(depth[u], u);
    for (int v : g[u]) {
        if (v == par) { continue; }
        depth[v] = depth[u] + 1;
        dfs(v, u);
        tour.emplace_back(depth[u], u);
    }
}
int lca(int u, int v) {
    u = pos[u], v = pos[v];
    if (u > v) { swap(u, v); }
    return rmq.get(u, v).second;
}
bool is_anc(int par, int u) { return lca(par, u) == par; }
int rooted_lca(int a, int b, int c) {
    return lca(a, b) ^ lca(b, c) ^ lca(c, a);
}
};
```

10.8.2 HLD

HLD.h, 74 lines

```
const int INF = 0x3f3f3f3f;
template<class SegmentTree>
struct HLD { // vertex update and max query on path u -> v
    int n;
    vector<vector<int>> g;
    SegmentTree seg_tree;
    vector<int> par, top, depth, sz, id;
    int timer = 0;
    bool VAL_IN_EDGE = false;
    HLD() {}
    HLD(int _n):
        n(_n), g(n), seg_tree(n), par(n), top(n), depth(n),
        sz(n), id(n) {}
    void build(int root = 0) {
        dfs_sz(root);
        dfs_hld(root);
    }
    void add_edge(int u, int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void dfs_sz(int u) {
        sz[u] = 1;
        for (int& v : g[u]) { // MUST BE ref for the swap
            below
            par[v] = u;
            depth[v] = depth[u] + 1;
            g[v].erase(find(g[v].begin(), g[v].end(), u));
            dfs_sz(v);
            sz[u] += sz[v];
            if (sz[v] > sz[g[u][0]]) { swap(v, g[u][0]); }
        }
    }
    void dfs_hld(int u) {
        id[u] = timer++;
        for (int v : g[u]) {
```

```
            top[v] = (v == g[u][0] ? top[u] : v);
            dfs_hld(v);
        }
    }
    int lca(int u, int v) {
        while (top[u] != top[v]) {
            if (depth[top[u]] > depth[top[v]]) { swap(u, v); }
            v = par[top[v]];
        }
        // now u, v is in the same heavy-chain
        return (depth[u] < depth[v] ? u : v);
    }
    int rooted_lca(int a, int b, int c) {
        return lca(a, b) ^ lca(b, c) ^ lca(c, a);
    }
    void set_vertex(int v, int x) { seg_tree.set(id[v], x); }
    void set_edge(int u, int v, int x) {
        if (u != par[v]) { swap(u, v); }
        seg_tree.set(id[v], x);
    }
    void set_subtree(int v, int x) {
        // modify segment_tree so that it supports range
        update
        seg_tree.set_range(
            id[v] + VAL_IN_EDGE, id[v] + sz[v] - 1, x);
    }
    int query_path(int u, int v) {
        int res = -INF;
        while (top[u] != top[v]) {
            if (depth[top[u]] > depth[top[v]]) { swap(u, v); }
            int cur = seg_tree.query(id[top[v]], id[v]);
            res = max(res, cur);
            v = par[top[v]];
        }
        if (depth[u] > depth[v]) { swap(u, v); }
        int cur = seg_tree.query(id[u] + VAL_IN_EDGE, id[v]);
        res = max(res, cur);
        return res;
    }
};
```

10.8.3 Centroid decomposition

Description: centroid decomposition technique for solving various task in a tree related to all paths/all pairs in tree.

Time: $O(N \log N)$

centroid_decomposition.h, 29 lines

```
const int N = (int) 1e5;
vector<int> g[N];
int sz[N];
bool vis[N];
void dfs_sz(int u, int par = -1) {
    sz[u] = 1;
    for (int v : g[u]) {
        if (v == par || vis[v]) { continue; }
        dfs_sz(v, u);
        sz[u] += sz[v];
    }
}
int find_cend(int u, int s, int par = -1) {
    for (int v : g[u]) {
        if (v == par || vis[v]) { continue; }
        if (sz[v] * 2 > s) { return find_cend(v, s, u); }
    }
    return u;
}
void solve(int u) {
```

```
    dfs_sz(u);
    int c = find_cend(u, sz[u]);
    vis[c] = true;
    // solve for vertex c...
    for (int v : g[c]) {
        if (vis[v]) { continue; }
        solve(v);
    }
}
```

10.8.4 DSU on tree

dsu_on_tree.h, 31 lines

```
const int nmax = (int) 2e5 + 1;
vector<int> adj[nmax];
int
    sz[nmax]; // sz[u] is the size of the subtree rooted at
    u
bool big[nmax];
void add(int u, int p, int del) {
    // do something...
    for (int v : adj[u]) {
        if (big[v] == false) { add(v, u, del); }
    }
}
void dsuOnTree(int u, int p, int keep) {
    int bigC = -1;
    for (int v : adj[u]) {
        if (v != p && (bigC == -1 || sz[bigC] < sz[v])) {
            bigC = v;
        }
    }
    for (int v : adj[u]) {
        if (v != p && v != bigC) { dsuOnTree(v, u, 0); }
    }
    if (bigC != -1) {
        big[bigC] = true;
        dsuOnTree(bigC, u, 1);
    }
    add(u, p, 1);
    if (bigC != -1) { big[bigC] = false; }
    if (keep == 0) { add(u, p, -1); }
}
```

10.8.5 Auxiliary tree

Description: building an auxiliary tree which contains vertices from the given vertex set of size N and their LCAs, there are at most $N - 1$ LCA vertices will be added, so the auxiliary tree will have at most $2N - 1$ vertices.

Time: $O(N \log N)$

auxiliary_tree.h, 27 lines

```
#include "<./lca.h>"
int build_tree(vector<vector<pair<int, int>>>& aux_g,
    vector<int>& vers, LCA& lca) {
    auto comp = [&](int u, int v) {
        return lca.pos[u] < lca.pos[v];
    };
    sort(vers.begin(), vers.end(), comp);
    for (int i = 0, sz = (int) vers.size(); i < sz - 1; ++i) {
        vers.emplace_back(lca.lca(vers[i], vers[i + 1]));
    }
    sort(vers.begin(), vers.end(), comp);
    vers.erase(unique(vers.begin(), vers.end()),
        vers.end());
    int aux_root = vers[0];
    vector<int> stack = {aux_root};
```



```

for (int i = 1; i < (int) vers.size(); ++i) {
    int u = vers[i];
    while (!stack.empty() && !lca.is_anc(stack.back(),
        u)) {
        stack.pop_back();
    }
    assert(!stack.empty());
    int w = lca.depth[u] - lca.depth[stack.back()];
    aux_g[stack.back()].push_back({u, w});
    stack.emplace_back(u);
}
return aux_root;
}

```

10.9 2-SAT

Description: finds a way to assign values to boolean variables a, b, c, \dots of a 2-SAT problem (each clause has at most two variables) so that the following formula becomes true: $(a \mid b) \& (\sim a \mid c) \& (b \mid \sim c) \dots$

Usage:

- TwoSat twosat(number of boolean variables);
- twosat.either(a, b); // a is true or b is false
- twosat.solve(); // return true iff the above formula is satisfiable

Time: $O(V + E)$ where V is the number of boolean variables and E is the number of clauses.

two_sat.h, 48 lines

```

struct TwoSat {
    int n;
    vector<vector<int>> g, tg; // g and transpose of g
    vector<int> comp, order;
    vector<bool> vis, vals;
    TwoSat(int _n):
        n(_n), g(2 * n), tg(2 * n), comp(2 * n), vis(2 * n),
        vals(n) {}
    void either(int u, int v) {
        u = max(2 * u, -2 * u - 1);
        v = max(2 * v, -2 * v - 1);
        g[u ^ 1].push_back(v);
        g[v ^ 1].push_back(u);
        tg[v].push_back(u ^ 1);
        tg[u].push_back(v ^ 1);
    }
    void set(int u) { either(u, u); }
    void dfs1(int u) {
        vis[u] = true;
        for (int v : g[u]) {
            if (!vis[v]) { dfs1(v); }
        }
        order.push_back(u);
    }
    void dfs2(int u, int scc_id) {
        comp[u] = scc_id;
        for (int v : tg[u]) {
            if (comp[v] == -1) { dfs2(v, scc_id); }
        }
    }
    bool solve() {
        for (int i = 0; i < 2 * n; ++i) {
            if (!vis[i]) { dfs1(i); }
        }
        fill(comp.begin(), comp.end(), -1);
        for (int i = 2 * n - 1, scc_id = 0; i >= 0; --i) {
            int u = order[i];
            if (comp[u] == -1) { dfs2(u, scc_id++); }
        }
        for (int i = 0; i < n; ++i) {
            int u = i * 2, nu = i * 2 + 1;

```

```

        if (comp[u] == comp[nu]) { return false; }
        vals[i] = comp[u] > comp[nu];
    }
    return true;
}
vector<bool> get_vals() { return vals; }
};

```

10.10 Manhattan MST

Description: given N points in the plane, the distance between two points is calculated as Manhattan distance. The function returns the list of edges which are guaranteed to contain a MST in the format (weight, u, v) of size up to $4N$

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

manhattan_mst.h, 36 lines

```

struct Point {
    int64_t x, y;
};
vector<tuple<int64_t, int, int>> manhattan_mst(
    vector<Point> ps) {
    vector<int> indices(ps.size());
    iota(indices.begin(), indices.end(), 0);
    vector<tuple<int64_t, int, int>> edges;
    for (int rot = 0; rot < 4; ++rot) {
        sort(indices.begin(), indices.end(), [&](int i, int
            j) {
                return (ps[i].x + ps[i].y < ps[j].x + ps[j].y);
            });
        map<int, int, greater<int>> active; // (xd, id)
        for (int i : indices) {
            for (auto it = active.lower_bound(ps[i].x);
                it != active.end(); active.erase(it++)) {
                int j = it->second;
                if (ps[i].x - ps[i].y > ps[j].x - ps[j].y) {
                    break;
                }
                assert(ps[i].x >= ps[j].x && ps[i].y >= ps[j].y);
                edges.emplace_back(
                    ps[i].x - ps[j].x + ps[i].y - ps[j].y, i, j);
            }
            active[ps[i].x] = i;
        }
        for (Point& p : ps) {
            if (rot & 1) {
                p.x *= -1;
            } else {
                swap(p.x, p.y);
            }
        }
        return edges;
    }
}

```

10.11 Matching

10.11.1 Kuhn algorithm

Description: Kuhn's algorithm for finding maximum matching in bipartite graph. For faster algorithm, see Hopcroft-Karp algorithm. g should be a list of neighbors of the left partition, $mat[v]$ will be the match for vertex v on the right partition, or -1 if no matching edge contains v .

Usage: vector<int> mat(right_sz, -1); bipartite_matching(g, mat);

Time: $O(VE)$

max_bipartite_matching_kuhn.h, 33 lines

```

int bipartite_matching(
    vector<vector<int>>& g, vector<int>& mat) {
    int n_left = (int) g.size();

```

```

    int n_right = (int) mat.size();
    vector<bool> used(n_left), pre_used(n_left);
    // finding some arbitrary matching to improve
    performance
    for (int u = 0; u < n_left; ++u) {
        for (int v : g[u]) {
            if (mat[v] == -1) {
                mat[v] = u;
                pre_used[u] = true;
                break;
            }
        }
    }
    auto find_aug_path = [&](auto&& self, int u) -> bool {
        if (used[u]) { return false; }
        used[u] = true;
        for (int v : g[u]) {
            if (mat[v] == -1 || self(self, mat[v])) {
                mat[v] = u;
                return true;
            }
        }
        return false;
    };
    for (int u = 0; u < n_left; ++u) {
        if (pre_used[u]) { continue; }
        fill(used.begin(), used.end(), false);
        find_aug_path(find_aug_path, u);
    }
    return n_right - count(mat.begin(), mat.end(), -1);
}

```

10.11.2 Hopcroft-Karp algorithm

Description: Hopcroft-Karp algorithm is a fast algorithm for finding maximum matching in bipartite graph. g should be a list of neighbors of the left partition.

Usage:

- vector<int> left_mat(n_left, -1), right_mat(n_right, -1);
- hopcroft_karp(g, left_mat, right_mat);

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

max_bipartite_matching_hopcroft_karp.h, 60 lines

```

int hopcroft_karp(vector<vector<int>>& g,
    vector<int>& left_mat, vector<int>& right_mat) {
    int n_left = (int) g.size();
    vector<int> dist(n_left);
    int matching = 0;

    auto bfs = [&]() {
        queue<int> que;
        for (int i = 0; i < (int) left_mat.size(); ++i) {
            if (left_mat[i] == -1) {
                dist[i] = 0;
                que.emplace(i);
            } else {
                dist[i] = -1;
            }
        }
        while (!que.empty()) {
            int u = que.front();
            que.pop();
            for (int v : g[u]) {
                if (right_mat[v] != -1 &&
                    dist[right_mat[v]] == -1) {
                    dist[right_mat[v]] = dist[u] + 1;
                    que.emplace(right_mat[v]);
                }
            }
        }
    };
}

```

```
};

auto dfs = [&](auto&& self, int u) {
    for (int v : g[u]) {
        if (right_mat[v] == -1) {
            left_mat[u] = v;
            right_mat[v] = u;
            return true;
        }
    }
    for (int v : g[u]) {
        if (dist[right_mat[v]] == dist[u] + 1) {
            if (self(self, right_mat[v])) {
                left_mat[u] = v, right_mat[v] = u;
                return true;
            }
        }
    }
    dist[u] = -1;
    return false;
};

while (true) {
    bfs();
    int augment = 0;
    for (int u = 0; u < n_left; ++u) {
        if (left_mat[u] == -1) { augment += dfs(dfs, u); }
    }
    if (!augment) { break; }
    matching += augment;
}
return matching;
}
```

10.11.3 Minimum vertex cover

Description: finding minimum vertex cover in a bipartite graph. The minimum vertex cover set and the maximum matching set have the same size. The complement of a vertex cover in any graph is an independent set.

min_vertex_cover.h, 34 lines

```
#include "<./max_bipartite_matching_kuhn.h">

vector<int> min_vertex_cover(
    vector<vector<int>>& g, int n_left, int n_right) {
    vector<int> mat(n_right, -1), cover;
    int max_matching = bipartite_matching(g, mat);
    vector<bool> in_cover(n_left), visited(n_right);
    for (int u : mat) {
        if (u != -1) { in_cover[u] = true; }
    }
    queue<int> que;
    for (int u = 0; u < n_left; ++u) {
        if (!in_cover[u]) { que.emplace(u); }
    }
    while (!que.empty()) {
        int u = que.front();
        que.pop();
        in_cover[u] = false;
        for (int v : g[u]) {
            if (!visited[v] && mat[v] != -1) {
                visited[v] = true;
                que.emplace(mat[v]);
            }
        }
    }
    for (int i = 0; i < n_left; ++i) {
        if (in_cover[i]) { cover.emplace_back(i); }
    }
    for (int i = 0; i < n_right; ++i) {
```

```
        if (visited[i]) { cover.emplace_back(i + n_left); }
    }
    assert((int) cover.size() == max_matching);
    return cover;
}
```

11 Misc.

11.1 Ternary search

Description: given an unimodal function $f(x)$, find the maximum/minimum of $f(x)$. Unimodal means the function strictly increases/decreases first, reaches a maximum/minimum (at a single point or over an interval), and then strictly decreases/increases.

ternary_search.h, 28 lines

```
const double eps = 1e-9;
template<typename T> inline T func(T x) { return x * x; }

// these two functions below find min, for find max:
// change
// '<' below to '>'.
double ternary_search(double l, double r) { // min
    while (r - l > eps) {
        double mid1 = l + (r - l) / 3;
        double mid2 = r - (r - l) / 3;
        if (func(mid1) < func(mid2)) {
            r = mid2;
        } else {
            l = mid1;
        }
    }
    return l;
}

int ternary_search(int l, int r) { // min
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (func(mid) < func(mid + 1)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }
    return l;
}
```

11.2 Gray code

Description: Gray code is a binary numeral system where two successive values differ in only one bit.

gray_code.h, 10 lines

```
int gray_code(int n) { return n ^ (n >> 1); }

int rev_gray_code(int code) {
    int n = 0;
    while (code > 0) {
        n ^= code;
        code >>= 1;
    }
    return n;
}
```

11.3 Matrix

matrix.h, 45 lines

```
const int MOD = (int) 1e9 + 7;
const long long SMOD = 1LL * MOD * MOD;
```

```
template<typename T> struct Matrix {
    int N, M;
    vector<vector<T>> mat;
    Matrix(int _N, int _M, T v = 0): N(_N), M(_M) {
        mat.assign(N, vector<T>(M, v));
    }
    static Matrix identity(int n) { // return identity matrix.
        Matrix I(n, n);
        for (int i = 0; i < n; ++i) { I[i][i] = 1; }
        return I;
    }
    vector<T>& operator[](int r) { return mat[r]; }
    const vector<T>& operator[](int r) const {
        return mat[r];
    }
    Matrix& operator*=(const Matrix& other) {
        assert(M == other.N); // [N x M] [other.N x other.M]
        Matrix res(N, other.M);
        for (int r = 0; r < N; ++r) {
            for (int c = 0; c < other.M; ++c) {
                long long sum = 0;
                for (int g = 0; g < M; ++g) {
                    sum += (long long) mat[r][g] * other[g][c];
                    if (sum >= SMOD) { sum -= SMOD; }
                }
                res[r][c] = sum % MOD;
            }
        }
        mat.swap(res.mat);
        return *this;
    }
    friend Matrix powmod(Matrix a, long long e) {
        assert(a.N == a.M);
        Matrix res = Matrix::identity(a.N);
        while (e > 0) {
            if (e & 1) { res *= a; }
            a *= a;
            e >>= 1;
        }
        return res;
    }
};
```

11.4 K-th order statistic

Description: finding the k-th smallest element in the array in linear time. The array should be shuffled before calling this function.

Time: $O(N)$

kth_order_statistic.h, 37 lines

```
template<typename T>
T kth_order_statistic(vector<T> arr, int k) {
    int n = (int) arr.size();
    k -= 1;
    for (int l = 0, r = n - 1; ) {
        if (r <= l + 1) {
            if (r == l + 1 && arr[r] < arr[l]) {
                swap(arr[l], arr[r]);
            }
            return arr[k];
        }
        int mid = l + (r - l) / 2;
        swap(arr[mid], arr[l + 1]);
        if (arr[l] > arr[r]) { swap(arr[l], arr[r]); }
        if (arr[l + 1] > arr[r]) { swap(arr[l + 1], arr[r]); }
        if (arr[l] > arr[l + 1]) { swap(arr[l], arr[l + 1]); }

        // performing division
```

```

// barrier is arr[l + 1], i.e. median among a[l], a[l
+
// 1], a[r]
int i = l + 1, j = r;
T pivot = arr[l + 1];
for (;;) {
    while (arr[++i] < pivot);
    while (arr[--j] > pivot);
    if (i > j) { break; }
    swap(arr[i], arr[j]);
}

// inserting the barrier
arr[l + 1] = arr[j];
arr[j] = pivot;

if (j >= k) { r = j - 1; }
if (j <= k) { l = i; }
}
}

```

11.5 LIS

Description: finding increasing subsequence of an array.

Time: $O(N \log N)$

lis.h, 29 lines

```

int lis(const vector<int>& arr, vector<int>& indices) {
    int n = (int) arr.size();
    vector<int> dp, idx;
    vector<int> trace(n, -1);
    for (int i = 0; i < n; ++i) {
        // change to lower_bound to get a strictly increasing
        // subsequence.
        int j =
            (int) (upper_bound(dp.begin(), dp.end(), arr[i]) -
                    dp.begin());
        if (j == (int) dp.size()) {
            dp.emplace_back(arr[i]);
            trace[i] = (idx.empty() ? -1 : idx.back());
            idx.emplace_back(i);
        } else {
            dp[j] = arr[i];
            trace[i] = (j > 0 ? idx[j - 1] : -1);
            idx[j] = i;
        }
    }
    int cur = idx.back();
    while (cur != -1) {
        indices.emplace_back(cur);
        cur = trace[cur];
    }
    reverse(indices.begin(), indices.end());
    assert(indices.size() == dp.size());
    return dp.size();
}

```

11.6 Others

11.6.1 Increase stack size

On Linux, use the command `$ulimit -s 268435456` to increase the stack size to 256MB. On Windows, add option `-Wl,-stack=268435456` when compiling with `gcc/g++`.

11.6.2 Stress-testing

stresstest.sh, 15 lines

```
#!/usr/bin/env bash
```

```

FLAGS="-std=c++17 -DLOCAL -D_GLIBCXX_DEBUG -Wall -Wextra
-pedantic"
g++ "$FLAGS" gen.cpp -o gen.out
g++ "$FLAGS" brute.cpp -o brute.out
g++ "$FLAGS" sol.cpp -o sol.out

num_tests="{1:-1000000}"
for (( i = 0; i < num_tests; ++i )); do
    ./gen.out > in
    ./brute.out < in > ans
    ./sol.out < in > out
    diff -Zb ans out || break
    echo "Test $i passed"
done

```