# Can Tho University
## CTU.NegativeZero

## Contents

# 1 Contest

## 1.1 C++

```cpp
#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());

const int MOD = (int) 1e9 + 7;
const int INF = 0x3f3f3f3f;

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

    return 0;
}
```

## 1.2 Debug

```cpp
#define debug(...) { string _s = #__VA_ARGS__; replace(begin(_s), end(_s),
    ',', ' '); stringstream _ss(_s); istream_iterator<string> _it(_ss);
    out_error(_it, __VA_ARGS__);}

void out_error(istream_iterator<string> it) { cerr << '\n'; }

template<typename T, typename ...Args>
void out_error(istream_iterator<string> it, T a, Args... args) {
    cerr << " [" << *it << " = " << a << "] ";
    out_error(++it, args...);
}

template<typename T, typename G> ostream& operator<<(ostream &os, const
    pair<T, G> &p) {
    return os << "(" << p.first << ", " << p.second << ")";
}

template<class Con, class = decltype(begin(declval<Con>()))>
typename enable_if<!is_same<Con, string>::value, ostream&>::type
operator<<(ostream& os, const Con& container) {
    os << "{";
    for (auto it = container.begin(); it != container.end(); ++it)
        os << (it == container.begin() ? "" : ", ") << *it;
    return os << "}";
}
```

## 1.3 Java

```java
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());
        }
    }
}
```

### 1.4 sublime-build

```
{
    "cmd": ["g++", "-std=c++17", "-fmax-errors=5", "-DLOCAL", "-Wall", "-
    Wextra", "-o", "${file_path}/${file_base_name}.out", "${file}"],
    "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
    "working_dir": "${file_path}",
    "selector": "source.cpp, source.c++"
}
```

### 1.5 .bashrc

```
alias cpp='g++ -std=c++17 -fmax-errors=5 -DLOCAL -Wall -Wextra'

#Stress-testing
function test {
  SOL=$1
  CHECKER=$2
  for i in {1..100};
    do
      ./gen.out > in && ./"$CHECKER.out" < in > ans && ./"$SOL.out" < in >
    out && diff -Z out ans && echo "Test $i passed!!" || break;
    done
}
```

## 2 Data structures

### 2.1 Sparse table

```
int st[MAXN][K + 1];
for (int i = 0; i < N; i++) {
    st[i][0] = f(array[i]);
}
for (int j = 1; j <= K; j++) {
    for (int i = 0; i + (1 << j) <= N; i++) {
        st[i][j] = f(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
    }
}
// Range Minimum Queries.
int lg[MAXN + 1];
lg[1] = 0;
for (int i = 2; i <= MAXN; i++) {
    lg[i] = lg[i / 2] + 1;
}
int j = lg[R - L + 1];
int minimum = min(st[L][j], st[R - (1 << j) + 1][j]);
// Range Sum Queries.
long long sum = 0;
for (int j = K; j >= 0; j--) {
    if ((1 << j) <= R - L + 1) {
        sum += st[L][j];
        L += 1 << j;
    }
}
```

### 2.2 Ordered set

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

template<typename key_type>
using set_t = tree<key_type, null_type, less<key_type>, rb_tree_tag,
    tree_order_statistics_node_update>;

void example() {
    vector<int> nums = {1, 2, 3, 5, 10};
    set_t<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

```
struct Dsu {
    int n;
    vector<int> par, sz;
    Dsu(int _n) : n(_n) {
        sz.resize(n, 1);
        par.resize(n);
        iota(par.begin(), par.end(), 0);
    }
    int find(int v) {
        // finding leader/parrent of set that contains the element v.
        // with {path compression optimization}.
        return (v == par[v] ? v : par[v] = find(par[v]));
    }
    bool same(int u, int v) {
        return find(u) == find(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]]].push_back(i);
        }
        return result;
    }
};
```

## 2.4 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].
 */
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);
    }
};
```

## 2.5 Efficient segment tree

```
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;
    }
};
```

## 2.6 Persistent lazy segment tree

```
struct Vertex {
    int l, r;
    long long val, lazy;
    bool has_changed = false;
    Vertex() {}
    Vertex(int _l, int _r, long long _val, int _lazy = 0) : l(_l), r(_r),
    val(_val), lazy(_lazy) {}
};
struct PerSegmentTree {
    vector<Vertex> tree;
    vector<int> root;
    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 left = build(arr, l, mid);
        int right = build(arr, mid + 1, r);
        tree.emplace_back(left, right, tree[left].val + tree[right].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].l, tree[x].r, 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 left = add(tree[x].l, l, mid, u, v, amt);
        int right = add(tree[x].r, mid + 1, r, u, v, amt);
        tree.emplace_back(left, right, tree[left].val + tree[right].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);
        return get_sum(tree[x].l, l, mid, u, v) + get_sum(tree[x].r, mid +
1, r, u, v);
    }
    void push(int x, int l, int mid, int r) {
        if (!tree[x].has_changed) return;
        Vertex left = tree[tree[x].l];
        Vertex right = tree[tree[x].r];
        tree.emplace_back(left);
        tree[x].l = tree.size() - 1;
        tree.emplace_back(right);
        tree[x].r = tree.size() - 1;

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

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

        tree[tree[x].l].has_changed = true;
        tree[tree[x].r].has_changed = true;
        tree[x].lazy = 0;
        tree[x].has_changed = false;
    }
};
```

## 2.7 Disjoint sparse table

```
/**
 * Description: range query on a static array.
```

```cpp
 * Time: O(1) per query.
 * Tested: stress-test.
 */
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;
    }
};
```

## 2.8 Fenwick tree

```cpp
using tree_type = long long;
struct FenwickTree {
    int n;
    vector<tree_type> fenw_coeff, fenw;
    FenwickTree() {}
    FenwickTree(int _n) : n(_n) {
        fenw_coeff.assign(n, 0); // fenwick tree with coefficient (n - i).
        fenw.assign(n, 0); // normal fenwick tree.
    }
    void build(const vector<int> &A) {
        assert((int) A.size() == n);
        vector<int> diff(n);
        diff[0] = A[0];
        for (int i = 1; i < n; ++i) {
            diff[i] = A[i] - A[i - 1];
        }
        fenw_coeff[0] = (long long) diff[0] * n;
        fenw[0] = diff[0];
        for (int i = 1; i < n; ++i) {
            fenw_coeff[i] = fenw_coeff[i - 1] + (long long) diff[i] * (n - i);
            fenw[i] = fenw[i - 1] + diff[i];
        }
        for (int i = n - 1; i >= 0; --i) {
            int j = (i & (i + 1)) - 1;
            if (j >= 0) {
                fenw_coeff[i] -= fenw_coeff[j];
                fenw[i] -= fenw[j];
            }
        }
    }
    void add(vector<tree_type> &fenw, int i, tree_type val) {
        while (i < n) {
            fenw[i] += val;
            i |= (i + 1);
        }
    }
    tree_type __prefix_sum(vector<tree_type> &fenw, int i) {
        tree_type res{};
        while (i >= 0) {
            res += fenw[i];
            i = (i & (i + 1)) - 1;
        }
        return res;
    }
    tree_type prefix_sum(int i) {
        return __prefix_sum(fenw_coeff, i) - __prefix_sum(fenw, i) * (n - i - 1);
    }
    void range_add(int l, int r, tree_type val) {
        add(fenw_coeff, l, (n - l) * val);
        add(fenw_coeff, r + 1, (n - r - 1) * (-val));
        add(fenw, l, val);
        add(fenw, r + 1, -val);
    }
    tree_type range_sum(int l, int r) {
        return prefix_sum(r) - prefix_sum(l - 1);
    }
};
```

## 2.9 Implicit treap

```cpp
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) {}
};
// Binary search tree + min-heap.
struct Treap {
    Node *root;
    Treap() : root(nullptr) {}
    int get_cnt(Node *n) { return n ? n->cnt : 0; }
    void upd_cnt(Node *&n) {
        if (n) n->cnt = get_cnt(n->left) + get_cnt(n->right) + 1;
    }
    void push_rev(Node *treap) {
        if (!treap || !treap->rev) return;
        treap->rev = false;
        swap(treap->left, treap->right);
        if (treap->left) treap->left->rev ^= true;
        if (treap->right) treap->right->rev ^= true;
    }
    pair<Node*, Node*> split(Node *treap, int x, int smaller = 0) {
        if (!treap) return {};
        push_rev(treap);
        int idx = smaller + get_cnt(treap->left); // implicit val.
        if (idx <= x) {
            auto pr = split(treap->right, x, idx + 1);
            treap->right = pr.first;
            upd_cnt(treap);
            return {treap, pr.second};
        }
        else {
            auto pl = split(treap->left, x, smaller);
            treap->left = pl.second;
            upd_cnt(treap);
            return {pl.first, treap};
        }
    }
    Node* merge(Node *l, Node *r) {
        push_rev(l); push_rev(r);
        if (!l || !r) return (l ? l : r);
        if (l->prior < r->prior) {
            l->right = merge(l->right, r);
            upd_cnt(l);
            return l;
        }
        else {
            r->left = merge(l, r->left);
            upd_cnt(r);
            return r;
        }
    }
    void insert(int pos, int val) {
        if (!root) {
            root = new Node(val);
            return;
        }
        Node *l, *m, *r;
        m = new Node(val);
        tie(l, r) = split(root, pos - 1);
        root = merge(l, merge(m, r));
    }
    void erase(int pos_l, int pos_r) {
        Node *l, *m, *r;
        tie(l, r) = split(root, pos_l - 1);
        tie(m, r) = split(r, pos_r - pos_l);
        root = merge(l, r);
    }
    void reverse(int pos_l, int pos_r) {
        Node *l, *m, *r;
        tie(l, r) = split(root, pos_l - 1);
        tie(m, r) = split(r, pos_r - pos_l);
        m->rev ^= true;
        root = merge(l, merge(m, r));
    }
    int query(int pos_l, int pos_r);
        // returns answer for corresponding types of query.
    void inorder(Node *n) {
        if (!n) return;
        push_rev(n);
        inorder(n->left);
        cout << n->val << ' ';
        inorder(n->right);
    }
    void print() {
        inorder(root);
        cout << '\n';
    }
};
```

# 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

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

### 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

$$n^a + n^{a+1} + \cdots + n^b = \frac{n^{b+1} - n^a}{n-1},\ n \neq 1$$
$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$
$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$
$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2$$
$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$
$$1^5 + 2^5 + 3^5 + \cdots + n^5 = \frac{n^2(n+1)^2(2n^2+2n-1)}{12}$$
$$1^6 + 2^6 + 3^6 + \cdots + n^6 = \frac{n(n+1)(2n+1)(3n^4+6n^3-3n+1)}{42}$$
$$1^7 + 2^7 + 3^7 + \cdots + n^7 = \frac{n^2(n+1)^2(3n^4+6n^3-n^2-4n+2)}{24}$$

# 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 substring
 *  s[0..i] which is also a suffix of this substring.
 * Time complexity: O(|S|).
 */
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 Counting occurrences of each prefix

```
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
}
```

## 4.3 Knuth–Morris–Pratt algorithm

```
/**
 * Searching for a substring in a string.
 * Time complexity: O(N + M).
 */
vector<int> KMP(const string &text, const string &pattern) {
    int n = (int) text.length();
    int m = (int) pattern.length();
    string s = pattern + '$' + text;
    vector<int> pi = prefix_function(s);
    vector<int> indices;
    for (int i = 0; i < (int) s.length(); ++i) {
        if (pi[i] == m) {
            indices.push_back(i - 2 * m);
        }
    }
    return indices;
}
```

## 4.4 Suffix array

```
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());
        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) {
            // counting sort.
            for (int i = 0; i < n; ++i) norder[i] = (sa[i] - k + n) % n;
            for (int i = 0; i < n; ++i) 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, nv = v + k;
                if (nu >= n) nu -= n;
                if (nv >= n) nv -= 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; i < n; ++i) rank[sa[i]] = i;
        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;
        }
        // Note: lcp[i] = longest common prefix(sa[i - 1], sa[i]).
    }
};
```

## 4.5 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 complexity: O(N).
 * Tested: https://judge.yosupo.jp/problem/enumerate_palindromes, stress-
   tested.
 */
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);
```

```cpp
        int l = 0, r = 0;
        for (int i = 0; i < n; ++i) {
            int mirror = l + r - i + z;
            d[z][i] = (i > r ? 0 : min(d[z][mirror], r - i));
            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;
}
```

## 4.6 Trie

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

## 4.7 Hashing

```cpp
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 n = (int) pw.size();
        if (n >= m) return;
        pw.resize(m);
        for (int i = n; 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 rng((unsigned int) chrono::steady_clock::now().time_since_epoch().
    count());
uint64_t Hash61::BASE = (MOD >> 2) + rng() % (MOD >> 1);
vector<uint64_t> Hash61::pw = vector<uint64_t>(1, 1);
```

# 5 Number Theory

## 5.1 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: $\phi(ab) = \phi(a) \cdot \phi(b) \cdot \dfrac{d}{\phi(d)}$.
  - With $n = p_1^{k_1} \cdot p_2^{k_2} \cdots p_m^{k_m}$:

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

- Application in Euler's theorem:
  - If $\gcd(a, M) = 1$, then:

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

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

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

## 5.2 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:

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

.

## 5.3 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 $\pi(n)$).

## 5.4 Wilson's theorem

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

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

## 5.5 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:

$$64 = 55 + 8 + 1$$
$$85 = 55 + 21 + 8 + 1$$

```
vector<int> zeckendoft_theorem(int n) {
    vector<int> fibs = {1, 1};
    int sz = 2;
    while (fibs.back() <= n) {
        fibs.push_back(fibs[sz - 1] + fibs[s - 2]);
        sz++:
    }
    fibs.pop_back();
    vector<int> nums;
    int p = sz - 1;
    while (n > 0) {
        if (n >= fibs[p]) {
            nums.push_back(fibs[p]);
            n -= fibs[p];
        }
        p--;
    }
    return nums;
}
```

## 5.6 Bitwise operation

- $a + b = (a \oplus b) + 2(a \,\&\, b)$
- $a \mid b = (a \oplus b) + (a \,\&\, b)$
- $a \,\&\, (b \oplus c) = (a \,\&\, b) \oplus (a \,\&\, c)$
- $a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$
- $a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$
- $a \mid (a \,\&\, b) = a$
- $a \,\&\, (a \mid b) = a$
- $n = 2^k \Leftrightarrow !(n \,\&\, (n-1)) = 1$
- $-a = \sim a + 1$
- $(4i) \oplus (4i{+}1) \oplus (4i{+}2) \oplus (4i{+}3) = 0$

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

```
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).
}
```

## 5.7 Pollard's rho algorithm

```
const int PRIME_MAX = (int) 4e4; // for handle numbers <= 1e9.
const int LIMIT = (int) 1e9;
vector<int> primes;

void linear_sieve(int n);
num_type mulmod(num_type a, num_type b, num_type mod);
num_type powmod(num_type a, num_type n, num_type mod);

bool miller_rabin(num_type a, num_type d, int s, num_type mod) {
    // mod - 1 = a ^ (d * 2^s).
    num_type x = powmod(a, d, mod);
    if (x == 1 || x == mod - 1) return true;
    for (int i = 1; i <= s - 1; ++i) {
        x = mulmod(x, x, mod);
        if (x == mod - 1) return true;
    }
    return false;
}
bool is_prime(num_type n, int ITERATION = 10) {
    if (n < 4) return (n == 2 || n == 3);
    if (n % 2 == 0 || n % 3 == 0) return false;
    num_type d = n - 1;
    int s = 0;
    while (d % 2 == 0) {
        d /= 2;
        s++;
    }
    for (int i = 0; i < ITERATION; ++i) {
        num_type a = (num_type) (rand() % (n - 2)) + 2;
        if (miller_rabin(a, d, s, n) == false) {
            return false;
        }
    }
    return true;
}
num_type f(num_type x, int c, num_type mod) { // f(x) = (x^2 + c) % mod.
    x = mulmod(x, x, mod);
    x += c;
    if (x >= mod) x -= mod;
    return x;
}
num_type pollard_rho(num_type n, int c) {
    // algorithm to find a random divisor of 'n'.
    // using random function: f(x) = (x^2 + c) % n.
    num_type x = 2, y = x, d;
    long long p = 1;
    int dist = 0;
```

```
    while (true) {
        y = f(y, c, n);
        dist++;
        d = __gcd(llabs(x - y), n);
        if (d > 1) break;
        if (dist == p) { dist = 0; p *= 2; x = y; }
    }
    return d;
}
void factorize(int n, vector<num_type> &factors);
void llfactorize(num_type n, vector<num_type> &factors) {
    if (n < 2) return;
    if (n < LIMIT) {
        factorize(n, factors);
        return;
    }
    if (is_prime(n)) {
        factors.emplace_back(n);
        return;
    }
    num_type d = n;
    for (int c = 2; d == n; c++) {
        d = pollard_rho(n, c);
    }
    llfactorize(d, factors);
    llfactorize(n / d, factors);
}
vector<num_type> gen_divisors(vector<pair<num_type, int>> &factors) {
    vector<num_type> divisors = {1};
    for (auto &x : factors) {
        int sz = (int) divisors.size();
        for (int i = 0; i < sz; ++i) {
            num_type 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.
}
```

## 5.8 Bitset sieve

```
/**
 * Description: sieve of eratosthenes for large n (up to 1e9).
 * Time and space (tested on codeforces):
 *   + For n = 1e8: ~200 ms, 6 MB.
 *   + For n = 1e9: ~4000 ms, 60 MB.
 */
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';
}
```

## 5.9 Block sieve

```
/**
 * Description: very fast sieve of eratosthenes for large n (up to 1e9).
 * Source: kactl.
 * Time and space (tested on codeforces):
 *   + For n = 1e8: ~160 ms, 60 MB.
 *   + For n = 1e9: ~1600 ms, 505 MB.
 * Need to check memory limit.
 */
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;
}
```

## 5.10 Combinatorics

### 5.10.1 Catalan numbers

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

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

- The first 12 Catalan numbers ($n = 0, 1, 2, \dots, 11$):

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786$$

- 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.

### 5.10.2 Stirling numbers of the second kind

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

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

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

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

### 5.10.3 Derangements

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

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

# 6 Linear algebra

## 6.1 Gauss elimination

```
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;
}
```

# 7 Geometry

## 7.1 Fundamentals

### 7.1.1 Point

```
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;}
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;
}
```

### 7.1.2 Line

```
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 1;
}
```

```
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;
    }
}
```

### 7.1.3 Circle

```
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;
}
```

### 7.1.4 Triangle

```
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));
}
```

### 7.1.5 Convex hull

```
vector<point> CH_Andrew(vector<point> &Pts) { // overall O(n log n)
    int n = Pts.size(), k = 0;
    vector<point> H(2 * n);
    sort(Pts.begin(), Pts.end());
    for (int i = 0; i < n; ++i) {
        while ((k >= 2) && !ccw(H[k - 2], H[k - 1], Pts[i])) --k;
        H[k++] = Pts[i];
    }
    for (int i = n - 2, t = k + 1; i >= 0; --i) {
        while ((k >= t) && !ccw(H[k - 2], H[k - 1], Pts[i])) --k;
        H[k++] = Pts[i];
    }
    H.resize(k);
    return H;
}
```

### 7.1.6 Polygon

```
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;
}
```

## 7.2 Minimum enclosing circle

```
/**
 * Description: computes the minimum circle that encloses all the given
    points.
 */
double abs(point a) { return sqrt(a.X * a.X + a.Y * a.Y); }

point center_from(double bx, double by, double cx, double cy) {
    double B = bx * bx + by * by, C = cx * cx + cy * cy, D = bx * cy - by *
     cx;
    return point((cy * B - by * C) / (2 * D), (bx * C - cx * B) / (2 * D));
}

circle circle_from(point A, point B, point C) {
    point I = center_from(B.X - A.X, B.Y - A.Y, C.X - A.X, C.Y - A.Y);
    return circle(I + A, abs(I));
}

const int N = 100005;
int n, x[N], y[N];
point a[N];

circle emo_welzl(int n, vector<point> T) {
    if (T.size() == 3 || n == 0) {
        if (T.size() == 0) return circle(point(0, 0), -1);
        if (T.size() == 1) return circle(T[0], 0);
        if (T.size() == 2) return circle((T[0] + T[1]) / 2, abs(T[0] - T
    [1]) / 2);
        return circle_from(T[0], T[1], T[2]);
    }
    random_shuffle(a + 1, a + n + 1);
    circle Result = emo_welzl(0, T);
    for (int i = 1; i <= n; i++)
        if (abs(Result.X - a[i]) > Result.Y + 1e-9) {
            T.push_back(a[i]);
            Result = emo_welzl(i - 1, T);
            T.pop_back();
        }
    return Result;
}
```

# 8 Graph

## 8.1 Bellman-Ford algorithm

```
/**
 * Description: single source shortest path in a weighted (negative or
    positive) directed graph.
 * Time: O(N * M).
 * Tested: https://open.kattis.com/problems/shortestpath3
 */
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) {}
};
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;
}
```

## 8.2 Articulation point and Bridge

```
/**
 * Description: finding articulation points and bridges in a simple
    undirected graph.
 * Tested: https://oj.vnoi.info/problem/graph_
 */
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) {
    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[v]) {
                bridges.emplace_back(u, v);
            }
            if (u != prev && low[v] >= num[u]) joint[u] = true;
        }
    }
    if (u == prev && child > 1) joint[u] = true;
}

int main() {
    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, i);
    }
    return 0;
}
```

## 8.3 Strongly connected components

```
/**
 * Description: Tarjan's algorithm finds strongly connected components
 *  in a directed graph. If vertices u and v belong to the same component,
 *  then scc_id[u] == scc_id[v].
 * Tested: https://judge.yosupo.jp/problem/scc
 */
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();
            debug(u, v)
            used[v] = true;
            scc_id[v] = scc;
        } while (v != u);
        scc++;
```

```
    }
}
```

## 8.4 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 (i.e, topo.size() < n).
 * Tested: https://judge.yosupo.jp/problem/scc
 */
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;
}
```

## 8.5 K-th smallest shortest path

```
/** Finding the k-th smallest shortest path from vertex s to vertex t,
 *  each vertex can be visited more than once.
 */
using adj_list = vector<vector<pair<int, int>>>;
vector<int> 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(ans.size() == k);
    return ans;
}
```

## 8.6 Eulerian path

### 8.6.1 Directed graph

```
/**
 * Hierholzer's algorithm.
 * Description: 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 complexity: O(|E|).
 */
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;
}
```

### 8.6.2 Undirected graph

```
/**
 * Hierholzer's algorithm.
 * Description: 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 complexity: O(|E|).
 */
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(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();
}
```

# 9 Misc.

## 9.1 Ternary search

```
const double eps = 1e-9;
double ternary_search_max(double l, double r) {
    // find x0 such that: f(x0) > f(x), \all x: l <= x <= r.
    while (r - l > eps) {
        double mid1 = l + (r - l) / 3;
        double mid2 = r - (r - l) / 3;
        if (f(mid1) < f(mid2)) l = mid1;
        else r = mid2;
    }
    return l;
}
double ternary_search_min(double l, double r) {
    // find x0 such that: f(x0) < f(x), \all x: l <= x <= r.
    while (r - l > eps) {
        double mid1 = l + (r - l) / 3;
        double mid2 = r - (r - l) / 3;
        if (f(mid1) > f(mid2)) l = mid1;
        else r = mid2;
    }
    return l;
}
```

## 9.2 Dutch flag national problem

```
void dutch_flag_national(vector<int> &arr) {
    // All elements that are LESS than pivot are moved to the LEFT.
    // All elements that are GREATER than pivot are moved to the RIGHT.
    // E.g. [1, 2, 0, 0, 2, 2, 1], pivot = 1 -> [0, 0, 1, 1, 2, 2, 2].
    int n = (int) arr.size();
    int i = 0, j = 0, k = n - 1;
    int pivot = 1;
    // 0....i....j....k....n
    while (j <= k) {
        if (arr[j] < pivot) {
            swap(arr[i], arr[j]);
            i++;
            j++;
        }
        else if (arr[j] > pivot) {
            swap(arr[j], arr[k]);
            k--;
        }
        else {
            j++;
        }
    }
    // 0 <= index <= i - 1: arr[index] < mid.
    // i <= index <= k: arr[index] = mid.
    // k + 1 <= index < sz: arr[index] > mid.
}
```

## 9.3 Matrix

```
struct Matrix {
    static const matrix_type INF = numeric_limits<matrix_type>::max();
    int N, M;
    vector<vector<matrix_type>> mat;

    Matrix(int _N, int _M, matrix_type v = 0) : N(_N), M(_M) {
        mat.assign(N, vector<matrix_type>(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<matrix_type>& operator[](int r) { return mat[r]; }
    const vector<matrix_type>& 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 square_mod = (long long) MOD * MOD;
                long long sum = 0;
                for (int g = 0; g < M; ++g) {
                    sum += (long long) mat[r][g] * other[g][c];
                    if (sum >= square_mod) sum -= square_mod;
                }
                res[r][c] = sum % MOD;
            }
```

```
        }
        mat.swap(res.mat); return *this;
    }
};
```